# Index Technicals

Team ENJOY!!

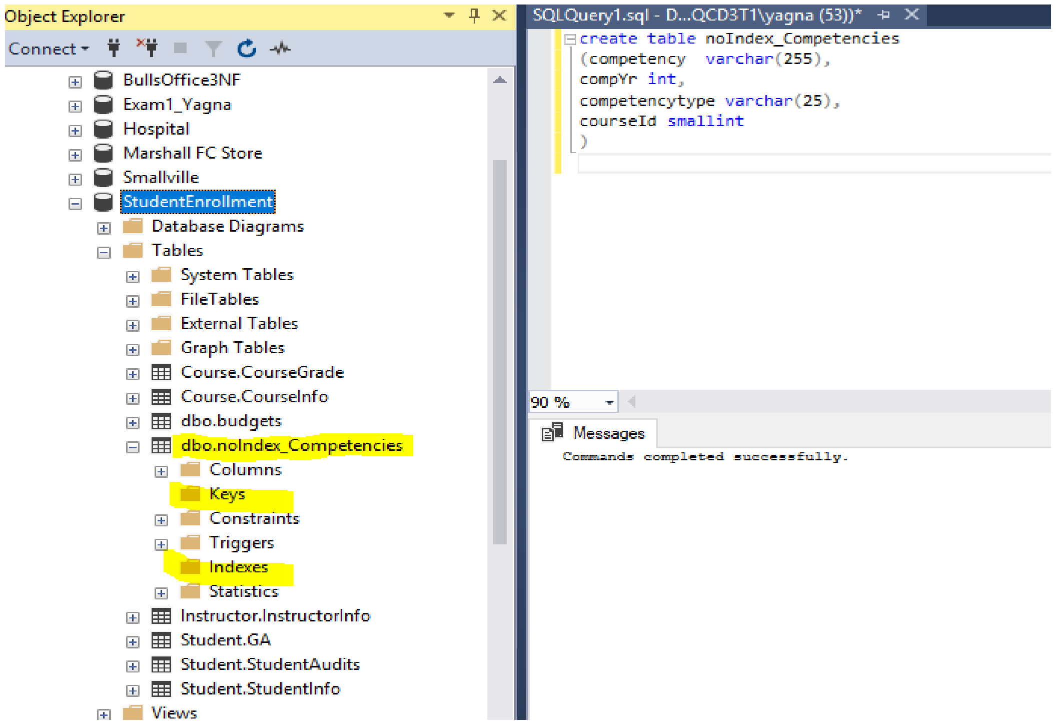
Assignment 10

Chukwuebeka Ezema
Neti Sheth
Josh Stamper
Yagna Venkitasamy

# TABLE OF CONTENTS

**<u>Create separate tables with the following configurations:</u>**

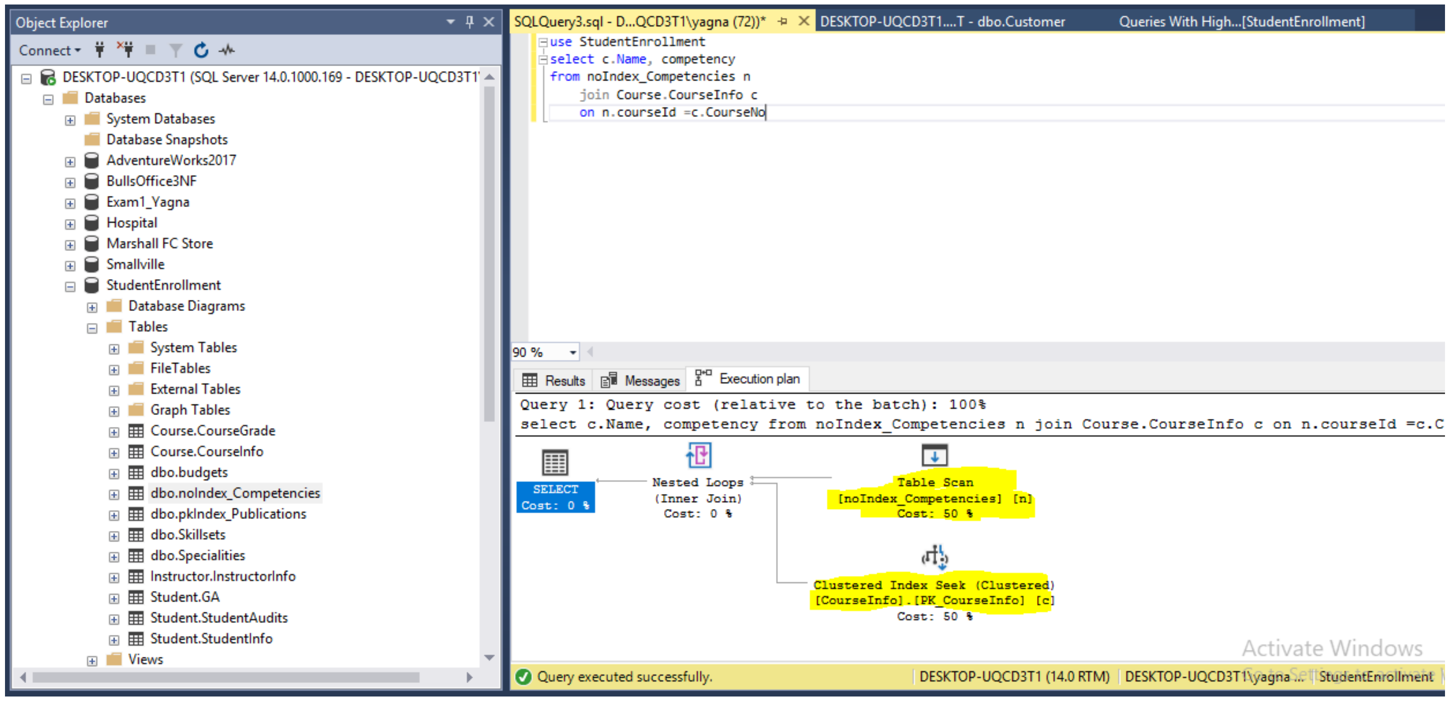
- **<u>No index</u>**
  Creating a Table with no index. The table is named “Competencies” that gives out a list of competencies that a student gains as part of enrolling in a course. This table has no primary key, hence no index whatsoever.



```
create table noIndex_Competencies
(competency  varchar(255),
compYr int,
competencytype varchar(25),
courseId smallint
)
```

```
Commands completed successfully.
```

With Execution Plan:



```
use StudentEnrollment
select c.Name, competency
from noIndex_Competencies n
    join Course.CourseInfo c
    on n.courseId =c.CourseNo
```

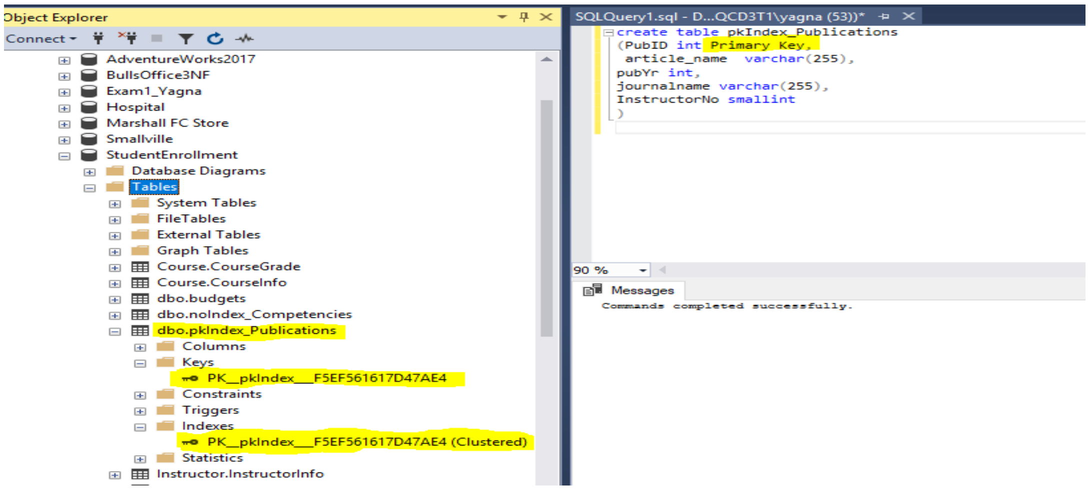
```
Query 1: Query cost (relative to the batch): 100%
select c.Name, competency from noIndex_Competencies n join Course.CourseInfo c on n.courseId =c.C
```

```
Query executed successfully.
```

We see that in the above query, the query cost is 50% each for the table scan and the index seek methods for running the query on the table with no index created.

- **PK index**
  Creating a Table with PK index. The table is named “Publications” that gives out a list of articles published by the instructor with the article name, published year and the Journal it was published. Publication ID is the primary key here that also serves as the primary key index.

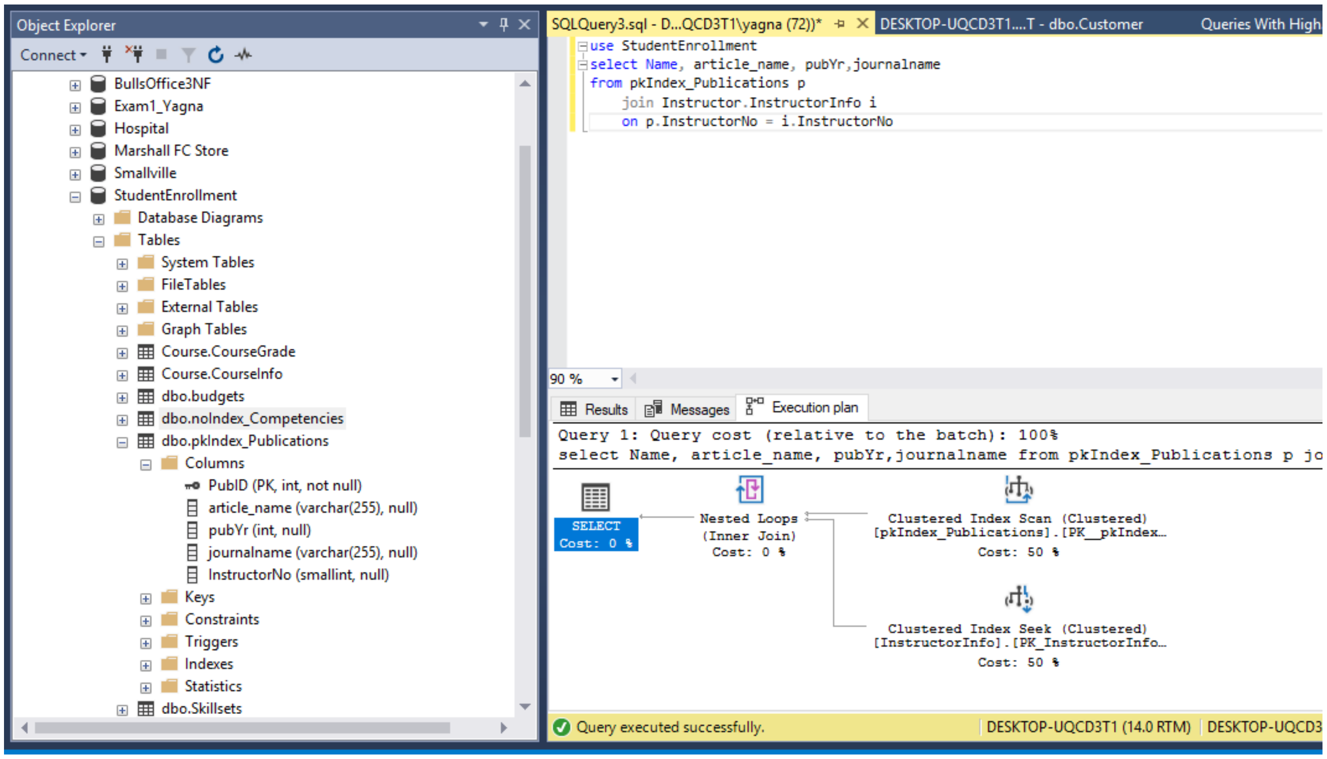
  The Primary Key is a logical object. By that we mean that is simply defines a set of properties on one column or a set of columns to require that the columns which make up the primary key are unique and that none of them are null. Because they are unique and not null, these values (or value if your primary key is a single column) can then be used to identify a single row in the table every time. In most of the database platforms the Primary Key will have an index auto-created on it.



```
create table pkIndex_Publications
(PubID int Primary Key,
 article_name  varchar(255),
pubYr int,
journalname varchar(255),
InstructorNo smallint
)
```

```
Commands completed successfully.
```

EXEC PLAN:



```
use StudentEnrollment
select Name, article_name, pubYr,journalname
from pkIndex_Publications p
    join Instructor.InstructorInfo i
    on p.InstructorNo = i.InstructorNo
```

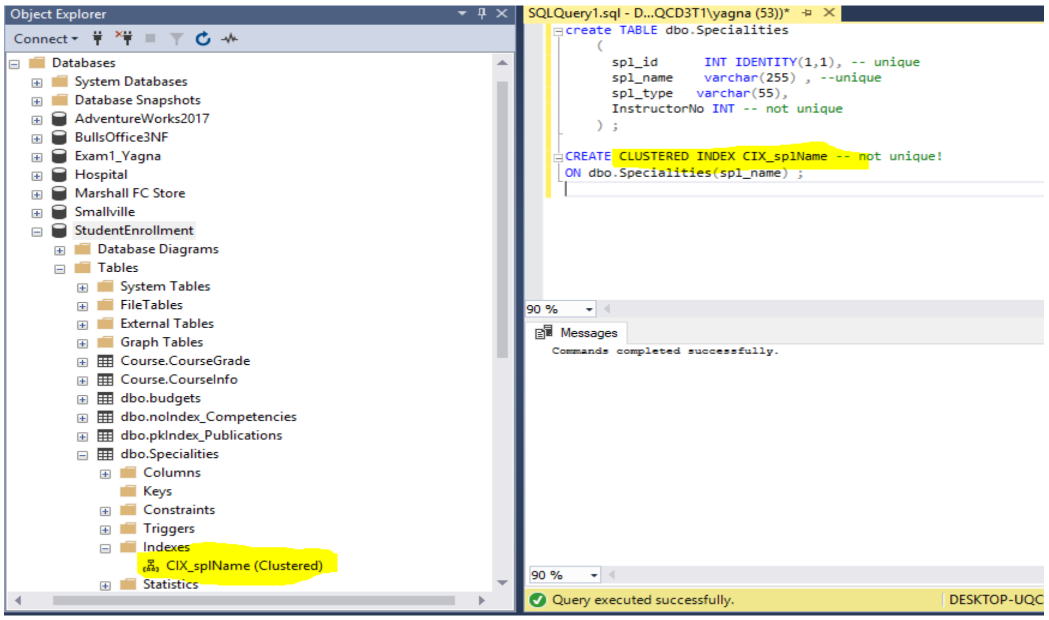
```
Query 1: Query cost (relative to the batch): 100%
select Name, article_name, pubYr,journalname from pkIndex_Publications p jo
```

We can see that using by using the primary key, a default index is added to the Publications table. The query uses the index scan method to scan through the primary key and uses Index seek method to get the InstructorNo from the Instructor table that matches the join query that is divided by 50:50 ratio.

- **Non PK clustered index:**

Primary key does not allow NULL, where in clustered index allow NULLs. Clustered index without primary key creates a Unique, PRIMARY KEY NONCLUSTERED index on the table. Primary key is a constraint and clustered index is an index so, logically they are both different entities. As a matter of fact, this is possible in one condition, when clustered index is created before the primary key on the table otherwise primary key will by default create the clustered index with it and only one clustered index is allowed per table.

In this example, we create a table with clustered index on non-primary key. The table name is Specialties that signifies the specialties possessed by the instructors and contains the columns Specialty ID, Specialty name, Specialty type and Instructor number which is a foreign key to the instructor table.

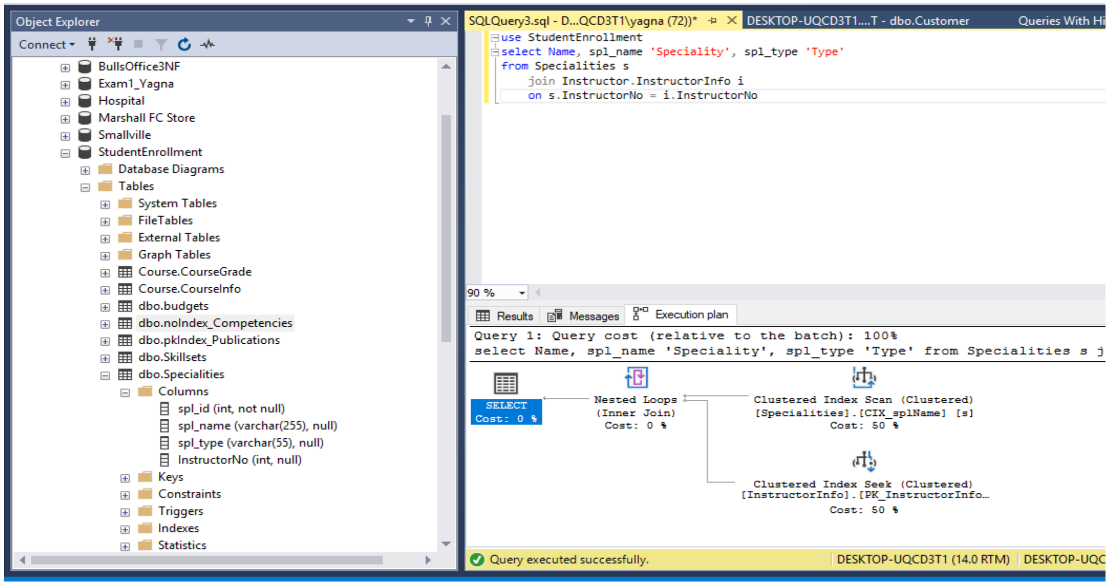


```
create TABLE dbo.Specialities
    (
      spl_id      INT IDENTITY(1,1), -- unique
      spl_name    varchar(255) , --unique
      spl_type   varchar(55),
      InstructorNo INT -- not unique
    ) ;

CREATE CLUSTERED INDEX CIX_splName -- not unique!
ON dbo.Specialities(spl_name) ;
```

```
Commands completed successfully.
```

Query executed successfully.

EXEC PLAN:



```
use StudentEnrollment
select Name, spl_name 'Speciality', spl_type 'Type'
from Specialities s
    join Instructor.InstructorInfo i
    on s.InstructorNo = i.InstructorNo
```

```
Query 1: Query cost (relative to the batch): 100%
select Name, spl_name 'Speciality', spl_type 'Type' from Specialities s j
```

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Clustered Index Scan (Clustered)
[Specialities].[CIX_splName] [s]
Cost: 50 %
Clustered Index Seek (Clustered)
[InstructorInfo].[PK_InstructorInfo...
Cost: 50 %

Query executed successfully.

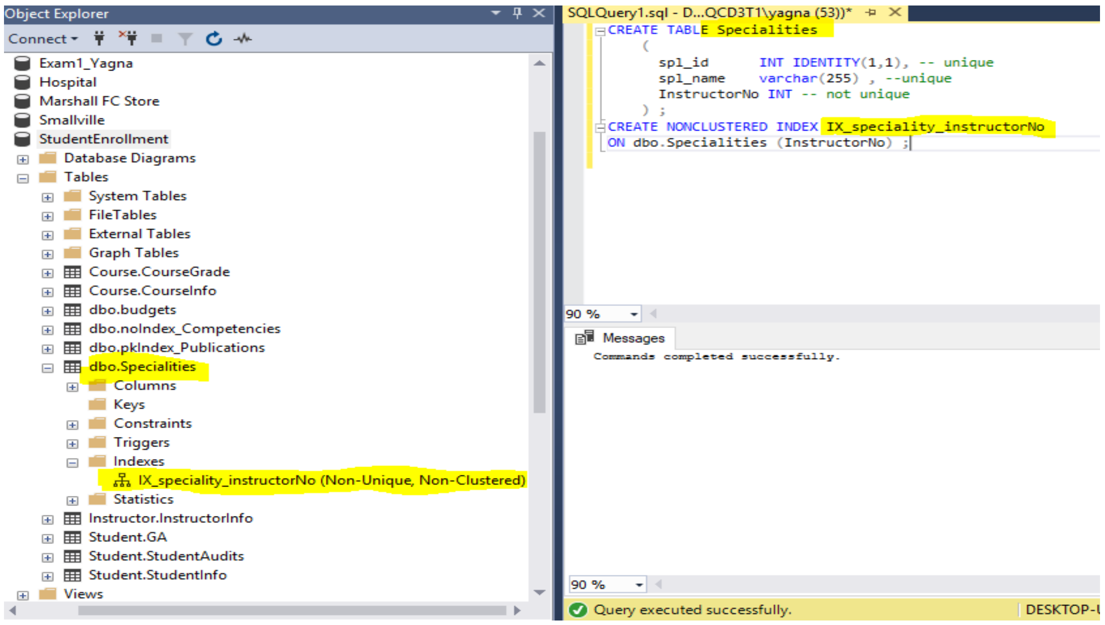
We can see that using even without the primary key, the clustered index is added to the Specialties table. The query uses the index scan method to scan through the Indexed column and uses Index seek method to get the InstructorNo from the Instructor table that matches the join query.

- **Only a Non-clustered index**

A non-clustered index is an index where the order of the rows does not match the physical order of the actual data. It is instead ordered by the columns that make up the index. In a non-clustered index, the leaf pages of the index do not contain any actual data, but instead contain pointers to the actual data. These pointers would point to the clustered index data page where the actual data exists (or the heap page if no clustered index exists on the table).

In this example, we are creating a non-clustered index in the same specialties table as we used above. The non-clustered index is created in the column Instructor number, which is not unique.

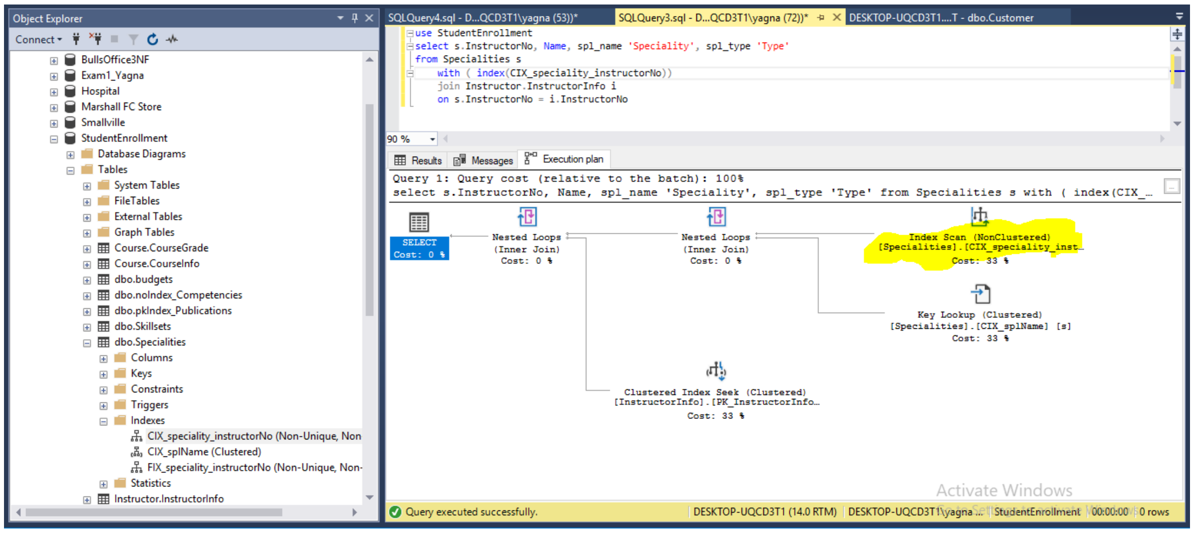
The main benefit to having a non-clustered index on a table is it provides fast access to data. The index allows the database engine to locate data quickly without having to scan through the entire table. As a table gets larger it is very important that the correct indexes are added to the table, as without any indexes query performance will drop off dramatically.



```
CREATE TABLE Specialities
    (
        spl_id        INT IDENTITY(1,1), -- unique
        spl_name      varchar(255) , --unique
        InstructorNo INT -- not unique
    ) ;
CREATE NONCLUSTERED INDEX IX_speciality_instructorNo
ON dbo.Specialities (InstructorNo) ;
```

```
Commands completed successfully.
```

Query executed successfully.



```
use StudentEnrollment
select s.InstructorNo, Name, spl_name 'Speciality', spl_type 'Type'
from Specialities s
    with ( index(CIX_speciality_instructorNo))
    join Instructor.InstructorInfo i
    on s.InstructorNo = i.InstructorNo
```

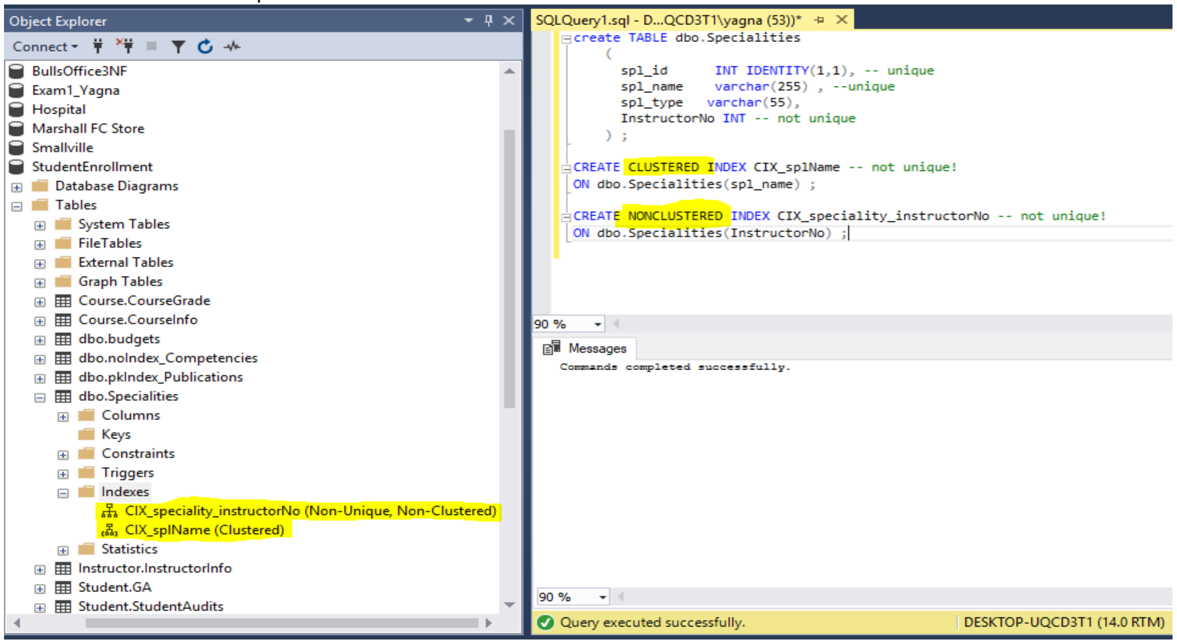
```
Query 1: Query cost (relative to the batch): 100%
select s.InstructorNo, Name, spl_name 'Speciality', spl_type 'Type' from Specialities s with ( index(CIX_…
```

Query executed successfully. | DESKTOP-UQCD3T1 (14.0 RTM) | DESKTOP-UQCD3T1\yagna ... | StudentEnrollment | 00:00:00 | 0 rows

We can see that using only a non-clustered index, the index is added to the Specialties table. The query uses the index scan method to scan through the non-clustered Indexed column and uses Index seek method to get the clustered indexed for the primary key InstructorNo from the Instructor table that matches the join query. The nested loop uses the key lookup method to seek the seek the clustered index column Specialty Name for the select query in the equal 33% cost for all 3.

- **Both Clustered Index and non-clustered Index:**

The clustered index is organized by the key columns. It also includes every other column as part of the row structure (i.e, it has the entire row). The non-clustered index is also organized by the key columns. It implicitly includes the clustering key columns (if the table is clustered), or a pointer to the row (if the table's a heap). If any INCLUDE columns are explicitly specified, they will also be included in the index structure.
There is a corner case where it makes sense to have a non-clustered index "duplicating" the clustered index. If we have a query that frequently scans the table, and ONLY makes use of the clustering key column, the query optimizer will prefer to use the non-clustered index. The non-clustered index does not contain the full row data, and thus it will take up less physical space. Because it takes up less physical space, SQL Server can scan the table with fewer IOs, and will make use of it for performance reasons.

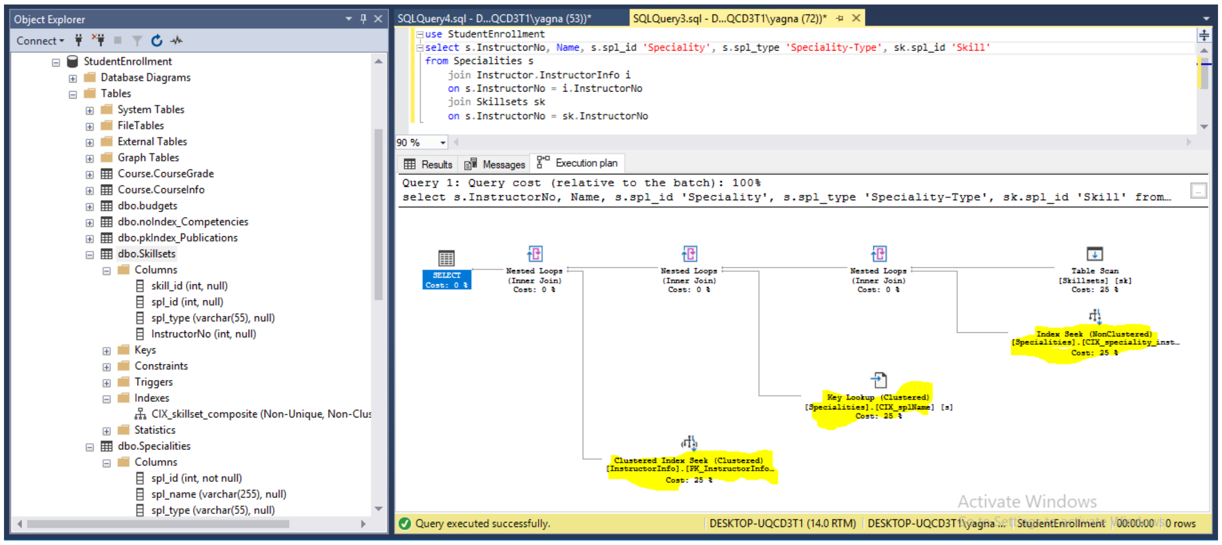


```
create TABLE dbo.Specialities
    (
      spl_id      INT IDENTITY(1,1), -- unique
      spl_name    varchar(255) , --unique
      spl_type    varchar(55),
      InstructorNo INT -- not unique
    ) ;

CREATE CLUSTERED INDEX CIX_splName -- not unique!
ON dbo.Specialities(spl_name) ;

CREATE NONCLUSTERED INDEX CIX_speciality_instructorNo -- not unique!
ON dbo.Specialities(InstructorNo) ;
```

```
Commands completed successfully.
```

Query executed successfully.

EXEC PLAN:



```
use StudentEnrollment
select s.InstructorNo, Name, s.spl_id 'Speciality', s.spl_type 'Speciality-Type', sk.spl_id 'Skill'
from Specialities s
    join Instructor.InstructorInfo i
    on s.InstructorNo = i.InstructorNo
    join Skillsets sk
    on s.InstructorNo = sk.InstructorNo
```

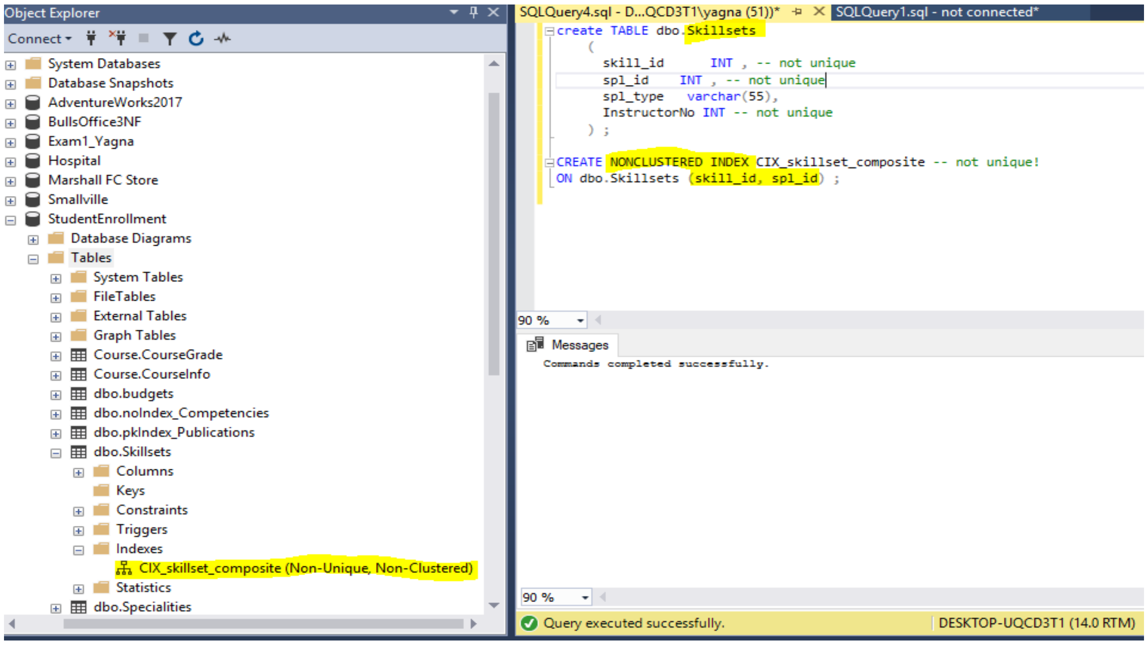
```
Query 1: Query cost (relative to the batch): 100%
select s.InstructorNo, Name, s.spl_id 'Speciality', s.spl_type 'Speciality-Type', sk.spl_id 'Skill' from…
```

Query executed successfully. | DESKTOP-UQCD3T1 (14.0 RTM) | DESKTOP-UQCD3T1\yagna ... | StudentEnrollment | 00:00:00 | 0 rows

We can see that using both clustered and non-clustered index, the clustered index is added to the Specialties table. The query uses the index seek method to scan through the non-clustered Indexed column and uses Index seek method to get the clustered indexed for the primary key InstructorNo from the Instructor table that matches the join query. The index uses the key lookup method to seek the PK clustered index for instructor no. The nested loop uses the key table scan method to seek skillsets table for the select query in the equal 25% cost for all 4.

- **Composite index (can only be non-clustered)**

An index that contains more than one column. In SQL Server, we can include up to 16 columns in an index, as long as the index doesn't exceed the 900-byte limit. Both clustered and non-clustered indexes can be composite indexes.

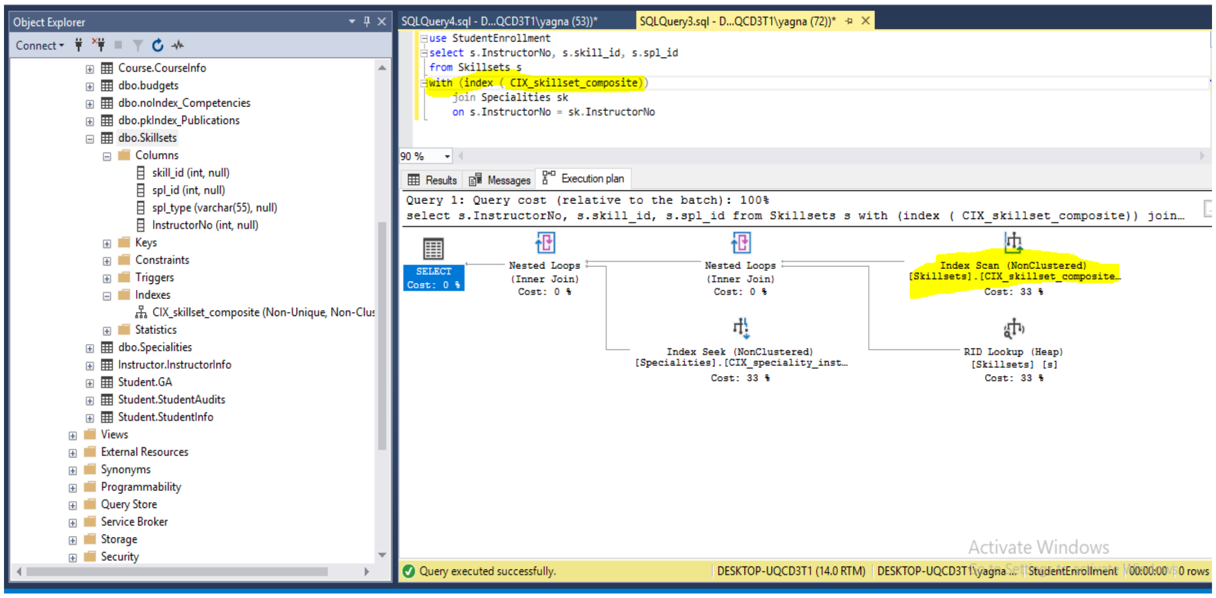


```
create TABLE dbo.Skillsets
    (
    skill_id      INT , -- not unique
    spl_id     INT , -- not unique
    spl_type   varchar(55),
    InstructorNo INT -- not unique
    ) ;

CREATE NONCLUSTERED INDEX CIX_skillset_composite -- not unique!
ON dbo.Skillsets (skill_id, spl_id) ;
```

```
Commands completed successfully.
```

```
Query executed successfully.    DESKTOP-UQCD3T1 (14.0 RTM)
```

EXEC PLAN:



```
use StudentEnrollment
select s.InstructorNo, s.skill_id, s.spl_id
from Skillsets s
with (index ( CIX_skillset_composite))
    join Specialities sk
    on s.InstructorNo = sk.InstructorNo
```

```
Query 1: Query cost (relative to the batch): 100%
select s.InstructorNo, s.skill_id, s.spl_id from Skillsets s with (index ( CIX_skillset_composite)) join…
```

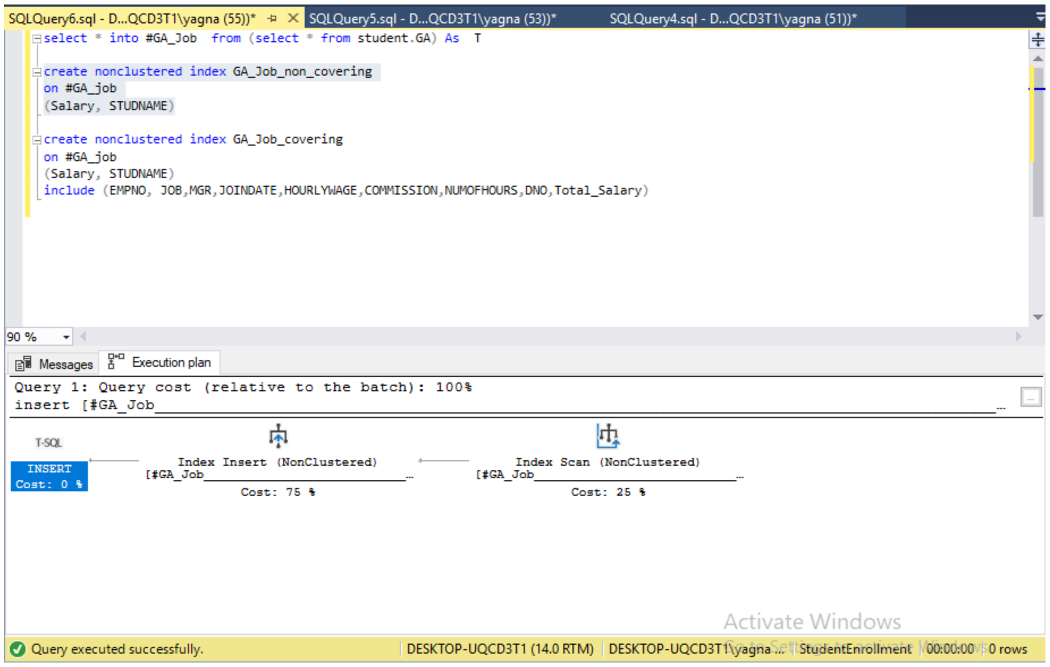
```
SELECT          Nested Loops     Nested Loops                      Index Scan (NonClustered)
Cost: 0 %       (Inner Join)     (Inner Join)                      [Skillsets].[CIX_skillset_composite…
                Cost: 0 %        Cost: 0 %                         Cost: 33 %

                   Index Seek (NonClustered)                       RID Lookup (Heap)
                   [Specialities].[CIX_speciality_inst…            [Skillsets] [s]
                   Cost: 33 %                                      Cost: 33 %
```

```
Query executed successfully.  DESKTOP-UQCD3T1 (14.0 RTM)  DESKTOP-UQCD3T1\yagna ...  StudentEnrollment  00:00:00  0 rows
```

We can see that by using a composite index, the index is added to the Skillsets table. The query uses the index scan method to scan through the non-clustered Indexed column for skillset table and uses Index seek method to get the non-clustered indexed for the Specialties ID. The nested loop uses the RID lookup also known as HEAP method to seek the composite index columns in the skillsets table for the select query in the equal 33% cost for all 3.

The query is using HEAP method as the composite indexes are non-clustered and the data stored in the table without a specified order.

- **Temp table with a covering and non-covering index:**
  An index that contains all information required to resolve the query is known as a “Covering Index”; it completely covers the query. In this we query, we create a temp table named GA_job and we create a covering and non-covering index in 2 different queries. The covering index query uses the Include syntax to include the columns in the table that will be covered in the indexing (that means all the columns in the table).

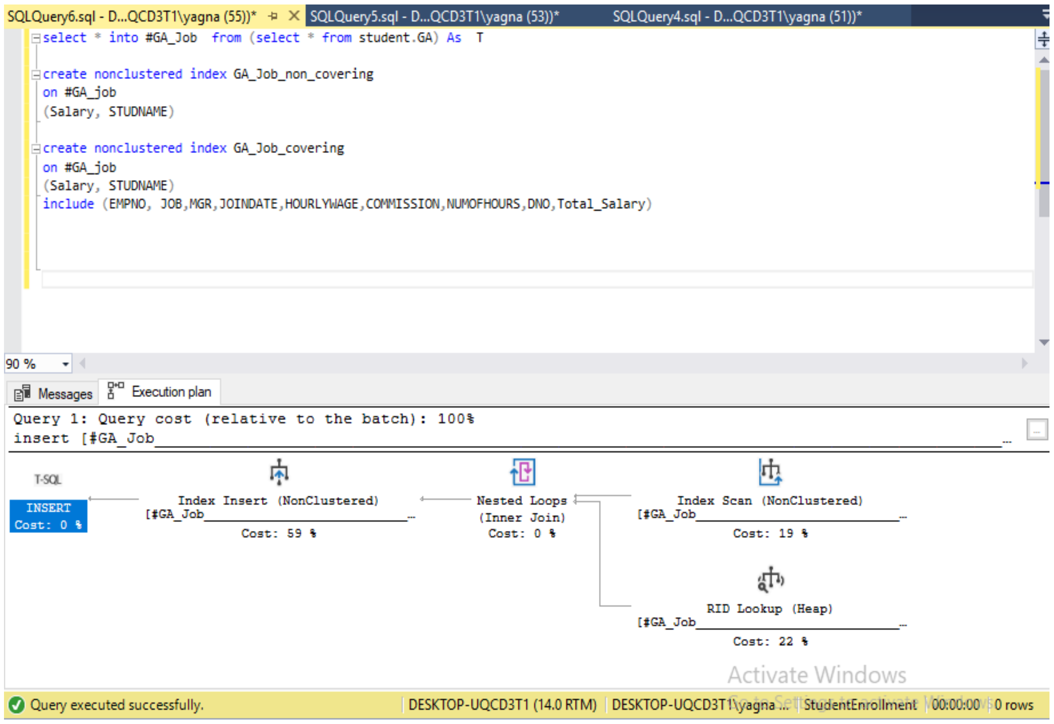


SQLQuery6.sql - D...QCD3T1\yagna (55))*  SQLQuery5.sql - D...QCD3T1\yagna (53))*  SQLQuery4.sql - D...QCD3T1\yagna (51))*

```
select * into #GA_Job  from (select * from student.GA) As  T

create nonclustered index GA_Job_non_covering
on #GA_job
(Salary, STUDNAME)

create nonclustered index GA_Job_covering
on #GA_job
(Salary, STUDNAME)
include (EMPNO, JOB,MGR,JOINDATE,HOURLYWAGE,COMMISSION,NUMOFHOURS,DNO,Total_Salary)
```

```
Query 1: Query cost (relative to the batch): 100%
insert [#GA_Job

T-SQL
INSERT        Index Insert (NonClustered)      Index Scan (NonClustered)
Cost: 0 %     [#GA_Job                         [#GA_Job
              Cost: 75 %                       Cost: 25 %
```

Query executed successfully. | DESKTOP-UQCD3T1 (14.0 RTM) | DESKTOP-UQCD3T1\yagna ... | StudentEnrollment | 00:00:00 | 0 rows



SQLQuery6.sql - D...QCD3T1\yagna (55))*  SQLQuery5.sql - D...QCD3T1\yagna (53))*  SQLQuery4.sql - D...QCD3T1\yagna (51))*

```
select * into #GA_Job  from (select * from student.GA) As  T

create nonclustered index GA_Job_non_covering
on #GA_job
(Salary, STUDNAME)

create nonclustered index GA_Job_covering
on #GA_job
(Salary, STUDNAME)
include (EMPNO, JOB,MGR,JOINDATE,HOURLYWAGE,COMMISSION,NUMOFHOURS,DNO,Total_Salary)
```

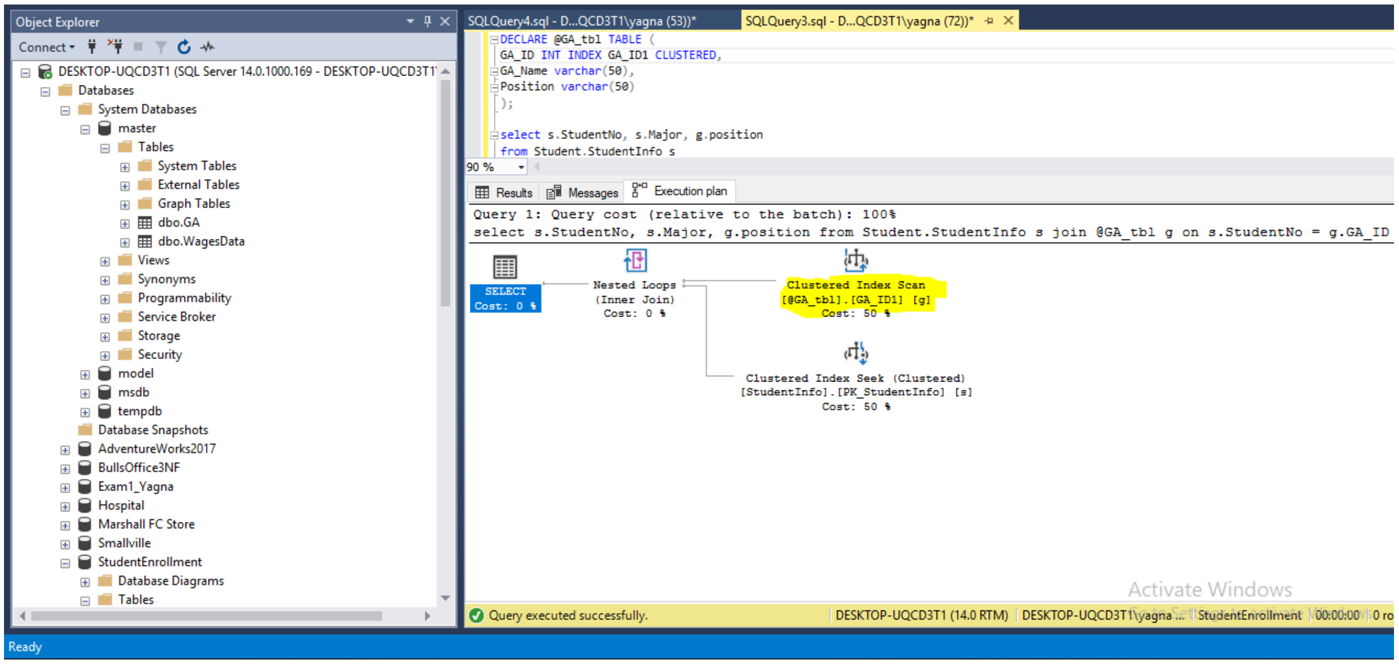
```
Query 1: Query cost (relative to the batch): 100%
insert [#GA_Job

T-SQL
INSERT        Index Insert (NonClustered)   Nested Loops     Index Scan (NonClustered)
Cost: 0 %     [#GA_Job                      (Inner Join)     [#GA_Job
              Cost: 59 %                    Cost: 0 %        Cost: 19 %

                                                             RID Lookup (Heap)
                                                             [#GA_Job
                                                             Cost: 22 %
```

Query executed successfully. | DESKTOP-UQCD3T1 (14.0 RTM) | DESKTOP-UQCD3T1\yagna ... | StudentEnrollment | 00:00:00 | 0 rows

The biggest benefit of covering index is that it contains all the fields required by query, it greatly improves the query performance and best results are achieved by placing fields required in the join criteria into the index key and placing fields required in the SELECT list into the INCLUDE part of the covering index. We can see that there is a greater benefit in using the covering index in the GA temp table. We see that the query cost is only 19% for the index scan for the non-clustered index and the Heap method cost is only 22% as compared to 25% for the initial index scan in the non-covering index.

- **<u>Table Variable with an index:</u>**

  Creating an index on a table variable can be done implicitly within the declaration of the table variable by defining a primary key and creating unique constraints. The primary key will represent a clustered index, while the unique constraint a non-clustered index. In this query, we are creating a table variable with an index on the GA table and running a query to join the student table to the GA table with a select statement.

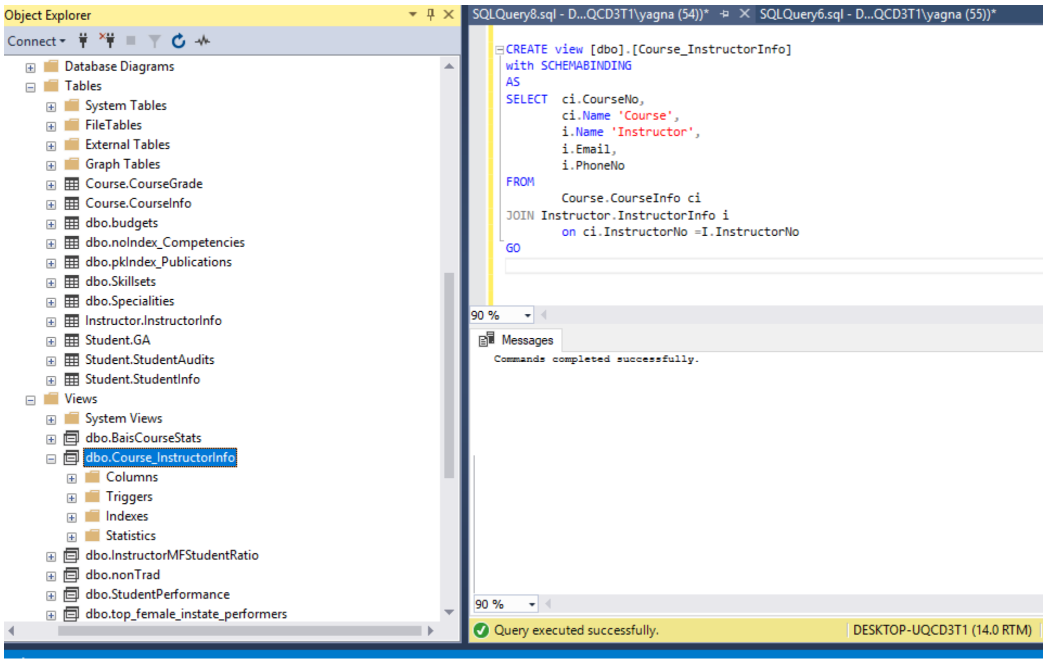


```sql
DECLARE @GA_tbl TABLE (
GA_ID INT INDEX GA_ID1 CLUSTERED,
GA_Name varchar(50),
Position varchar(50)
);

select s.StudentNo, s.Major, g.position
from Student.StudentInfo s
```

```
Query 1: Query cost (relative to the batch): 100%
select s.StudentNo, s.Major, g.position from Student.StudentInfo s join @GA_tbl g on s.StudentNo = g.GA_ID
```

Temp tables perform better in situations where an index is needed. The con to temp tables is that they will frequently cause recompilation. SQL server perform compilation at the statement level so if only one statement utilizes a temp table then that statement is the only one that gets recompiled.

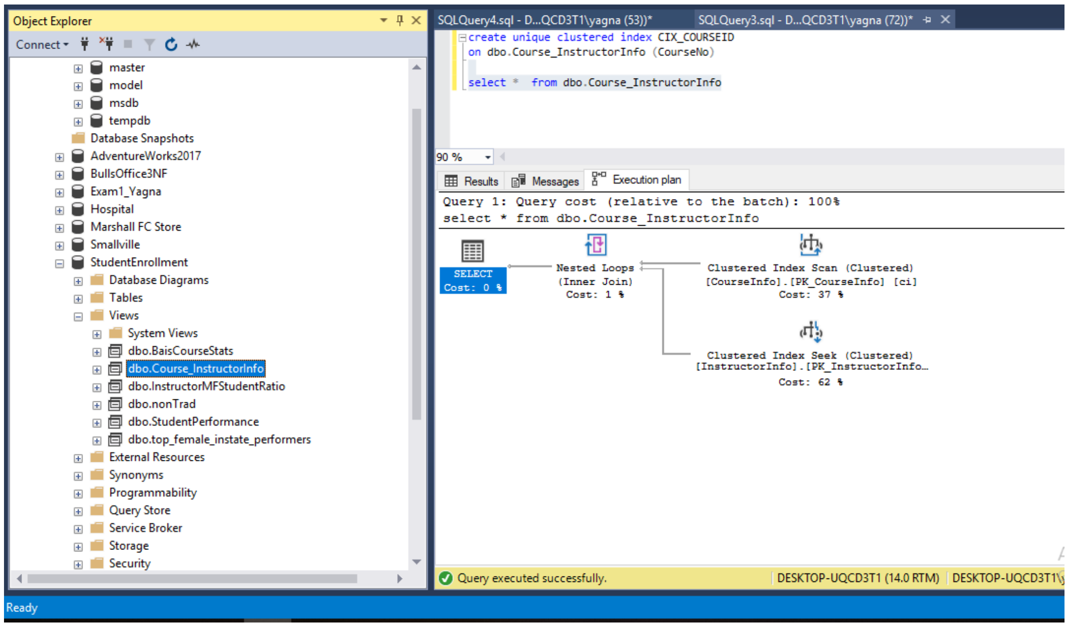
- **<u>Create a view with and without an index:</u>**
  Views are virtual tables that are used to retrieve a set of data from one or more tables. The view's data is not stored in the database, but the real retrieval of data is from the source tables. When you call the view, the source table's definition is substituted in the main query and the execution will be like reading from these tables directly.
  To enhance the performance of such complex queries, a unique clustered index can be created on the view, where the result set of that view will be stored in your database the same as a real table with a unique clustered index. The good thing here is – the queries that are using the table itself can benefits from the view's clustered index without calling the view itself.
  In this, we are creating a view on the table instructor info with SCHEMABINDING from the course info table with a join from the instructor table but no index in the view.



```
CREATE view [dbo].[Course_InstructorInfo]
with SCHEMABINDING
AS
SELECT  ci.CourseNo,
        ci.Name 'Course',
        i.Name 'Instructor',
        i.Email,
        i.PhoneNo
FROM
        Course.CourseInfo ci
JOIN Instructor.InstructorInfo i
        on ci.InstructorNo =I.InstructorNo
GO
```

```
Commands completed successfully.
```

Query executed successfully.



```
create unique clustered index CIX_COURSEID
on dbo.Course_InstructorInfo (CourseNo)

select *  from dbo.Course_InstructorInfo
```

```
Query 1: Query cost (relative to the batch): 100%
select * from dbo.Course_InstructorInfo
```

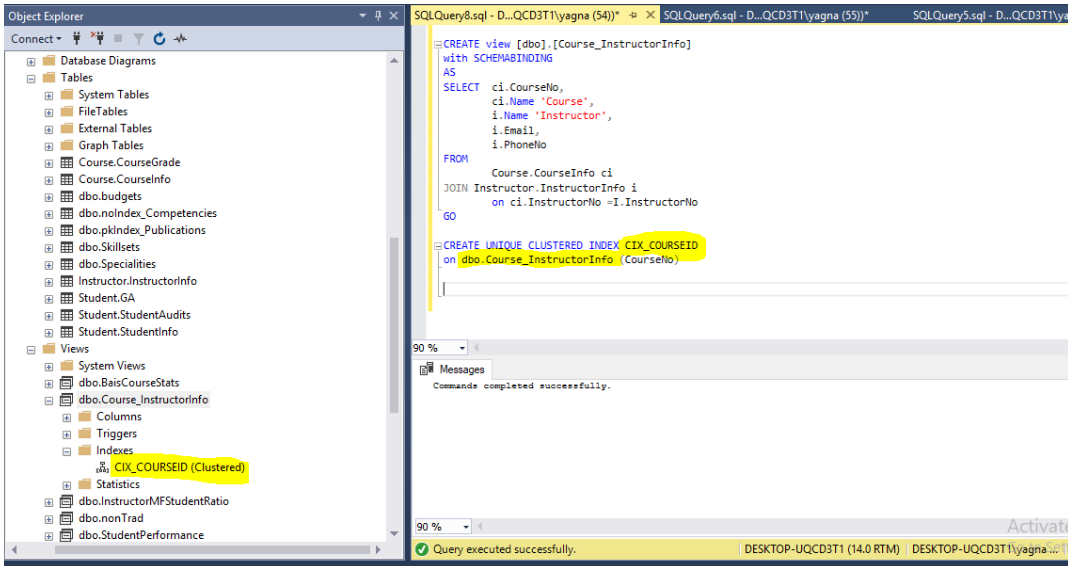
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 1 %
Clustered Index Scan (Clustered)
[CourseInfo].[PK_CourseInfo] [ci]
Cost: 37 %
Clustered Index Seek (Clustered)
[InstructorInfo].[PK_InstructorInfo…
Cost: 62 %

Query executed successfully.

We can see that by creating the view with no index the query utilizes the clustered indexes from the joined tables' primary keys and uses the clustered index scan and index seek with the query cost of 37% and 62%.

## **Execution of the view with an INDEX:**

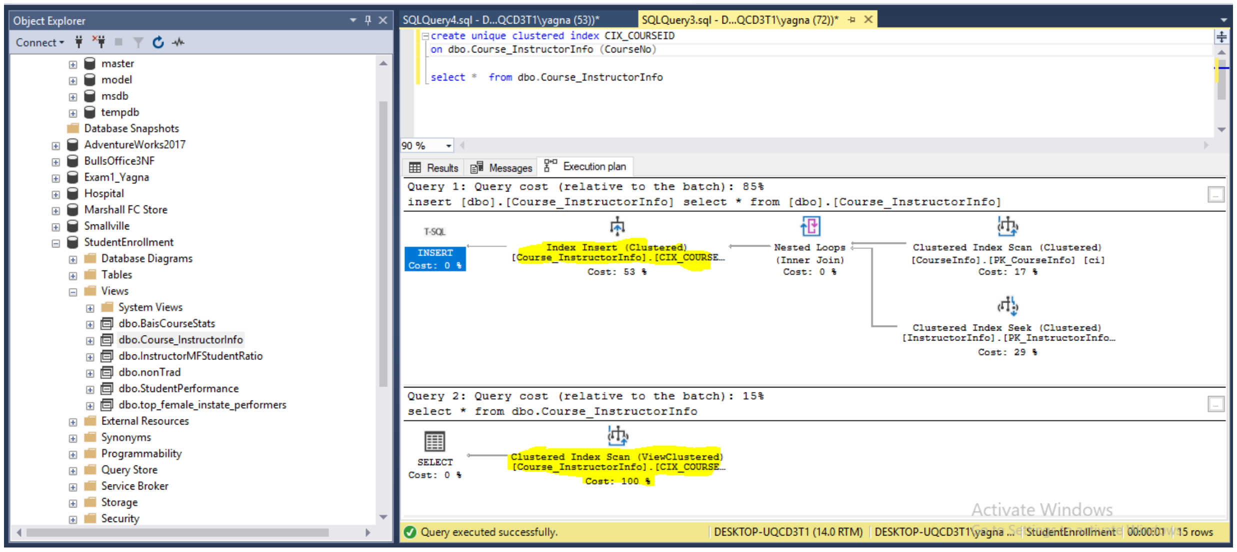
Any user-defined function that is referenced by the created indexed view should be created using WITH SCHEMABINDING hint. Once the Indexed view is created, its data will be stored in your database the same as any other clustered index, so the storage space for the view's clustered index should be taken into consideration. Having the indexed view's clustered index stored in the database, with its own statistics created to optimize the cardinality estimation, different from the underlying tables' statistics, the SQL engine will not waste the time substituting the source tables' definition in the main query, and it will read directly from the view's clustered index.



```
CREATE view [dbo].[Course_InstructorInfo]
with SCHEMABINDING
AS
SELECT  ci.CourseNo,
        ci.Name 'Course',
        i.Name 'Instructor',
        i.Email,
        i.PhoneNo
FROM
        Course.CourseInfo ci
JOIN Instructor.InstructorInfo i
        on ci.InstructorNo =I.InstructorNo
GO

CREATE UNIQUE CLUSTERED INDEX CIX_COURSEID
on dbo.Course_InstructorInfo (CourseNo)
```

```
Commands completed successfully.
```

Query executed successfully.



```
create unique clustered index CIX_COURSEID
on dbo.Course_InstructorInfo (CourseNo)

select *  from dbo.Course_InstructorInfo
```

```
Query 1: Query cost (relative to the batch): 85%
insert [dbo].[Course_InstructorInfo] select * from [dbo].[Course_InstructorInfo]

INSERT          Index Insert (Clustered)                     Nested Loops    Clustered Index Scan (Clustered)
Cost: 0 %       [Course_InstructorInfo].[CIX_COURSE...       (Inner Join)    [CourseInfo].[PK_CourseInfo] [ci]
                Cost: 53 %                                   Cost: 0 %       Cost: 17 %

                                                                             Clustered Index Seek (Clustered)
                                                                             [InstructorInfo].[PK_InstructorInfo...
                                                                             Cost: 29 %

Query 2: Query cost (relative to the batch): 15%
select * from dbo.Course_InstructorInfo

SELECT          Clustered Index Scan (ViewClustered)
Cost: 0 %       [Course_InstructorInfo].[CIX_COURSE...
                Cost: 100 %
```

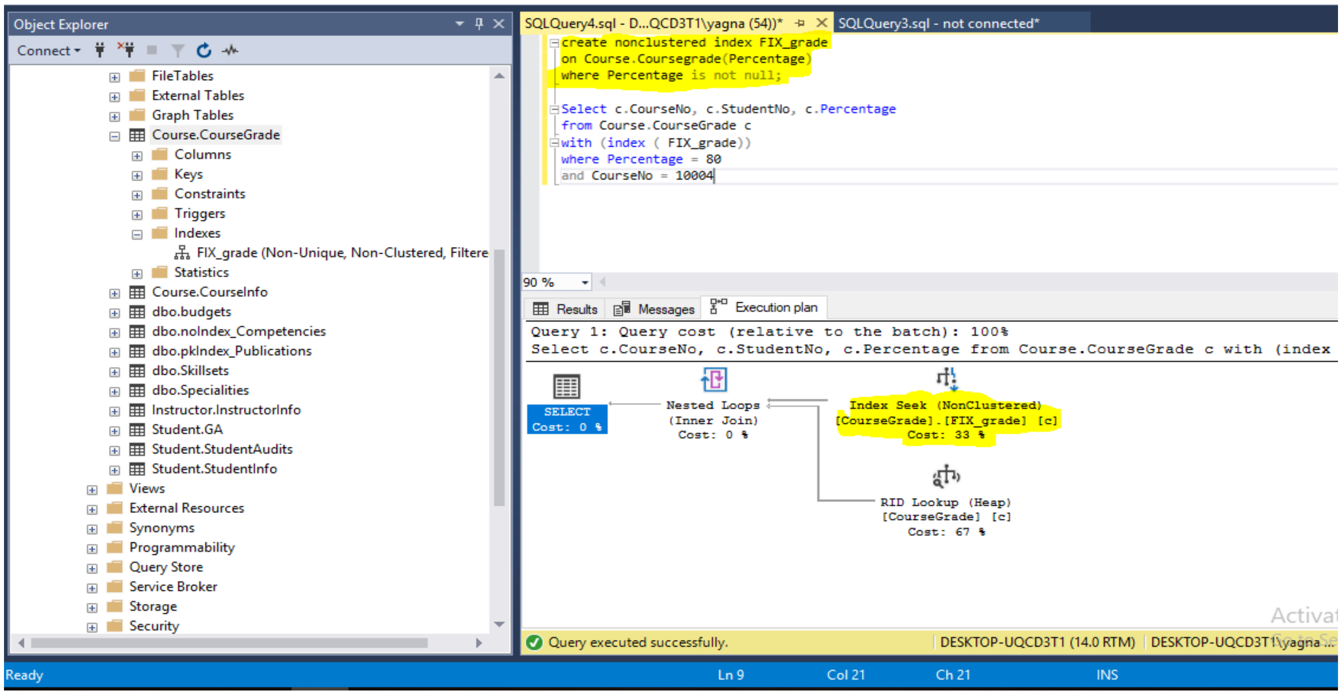
Query executed successfully.

We can see that by creating the view with the unique clustered index the query utilizes the clustered indexes from the joined tables' primary keys and uses the clustered index scan and index seek with the reduced query cost of 17% and 29%.

- **Filtered Index:**

  Filtered indexes are non-clustered indexes that have the addition of a WHERE clause. Although the WHERE clause in a filtered index allows only simple predicates, it provides notable improvements over a traditional non-clustered index. This allows the index to target specific data values – only records with certain values will be added to the index – resulting in a much smaller, more accurate, and more efficient index.

  Here, we add a filtered index to the percentage column of the coursegrade table. The query is run with the select statement with selected percentage and the courseNo forced with the Filtered Index using the WITH clause.



```
create nonclustered index FIX_grade
on Course.Coursegrade(Percentage)
where Percentage is not null;

Select c.CourseNo, c.StudentNo, c.Percentage
from Course.CourseGrade c
with (index ( FIX_grade))
where Percentage = 80
and CourseNo = 10004
```

```
Query 1: Query cost (relative to the batch): 100%
Select c.CourseNo, c.StudentNo, c.Percentage from Course.CourseGrade c with (index
```

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[CourseGrade].[FIX_grade] [c]
Cost: 33 %
RID Lookup (Heap)
[CourseGrade] [c]
Cost: 67 %
Object Explorer
FIX_grade (Non-Unique, Non-Clustered, Filtere
Query executed successfully.

We can see that the index seek method for the filtered index costs only 33% and the rest of it goes to the HEAP method for the non-clustered indexes. If we look at the current execution plan for this query, we see that there are no existing indexes that could be used for an index seek to locate the matching records efficiently, and that a clustered index scan was performed instead. An index seek operation happens when the query engine can use an appropriate index to find the exact location of the requested data, using the b-tree method. Some key benefits of using filtered indexes are:

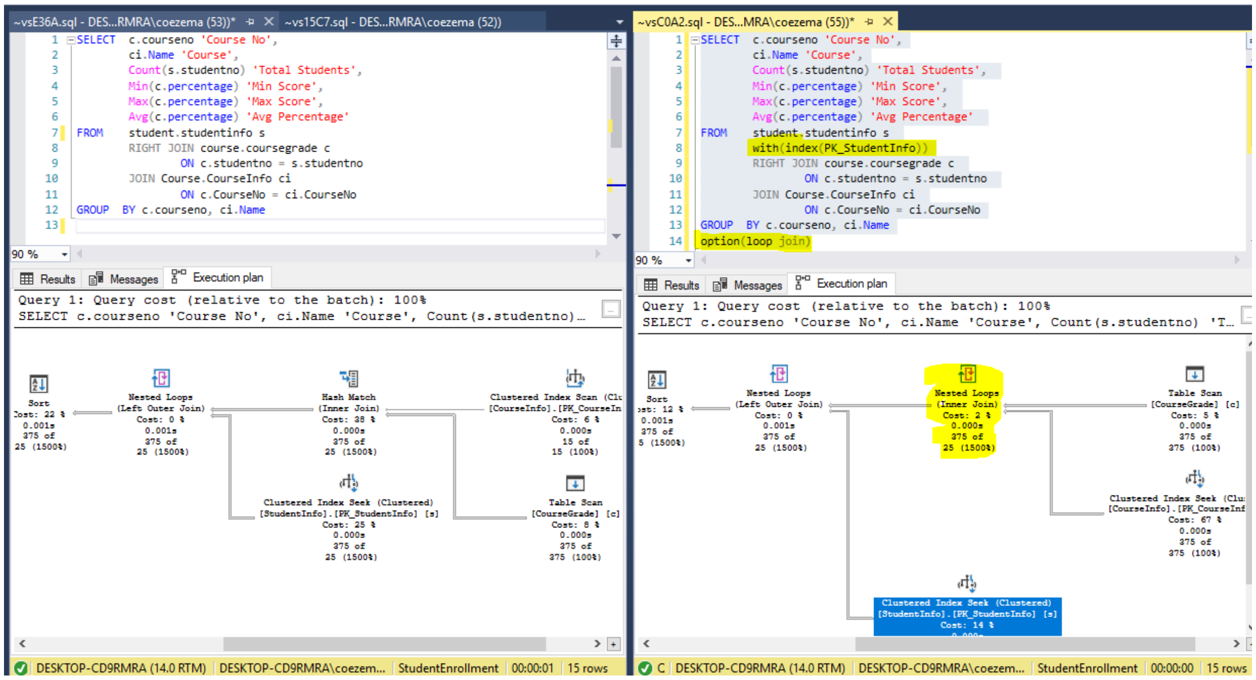
- **Reduced index maintenance costs**. Insert, update, delete, and merge operations are not as expensive, since a filtered index is smaller and does not require as much time for reorganization or rebuilding.
- **Reduced storage cost**. The smaller size of a filtered index results in a lower overall index storage requirement.
- **More accurate statistics**. Filtered index statistics cover only the rows the meet the WHERE criteria, so in general, they are more accurate than full-table statistics.
- **Optimized query performance**. Because filtered indexes are smaller and have more accurate statistics, queries and execution plans are more efficient.

  **Some Disadvantages include**:

- **Statistics may not get updated often enough**, depending on how often filtered column data is changed. Because of how SQL Server decides when to update statistics (when about 20% of a column's data has been modified), statistics could become quite out of date.
- **Filtered indexes cannot be created on views**. However, a filtered index on the base table of a view will still optimize a view's query.
- **XML indexes and full-text indexes cannot be filtered.** Only non-clustered indexes are able to take advantage of the WHERE clause.

**QUERY HINTS WITH LOOP JOIN AND WITH INDEX & Analyze performance of Index Choices and Choices of Hints:**

One of the things that the SQL Server query optimizer does is determine how to retrieve the data requested by our query. Sometimes the query optimizer has a lapse in judgement and creates a less-than-efficient plan, requiring us to step in. One way to “fix” a poor performing plan is to use an index hint. While we normally have no control over how SQL Server retrieves the data we requested, an index hint forces the query optimizer to use the index specified in the hint to retrieve the data. In this query, we are performing a comparison of the query hint to normal querying. We are calculating the Total students of a course and the score range in the course grade for the Courses table.



```
SELECT  c.courseno 'Course No',
        ci.Name 'Course',
        Count(s.studentno) 'Total Students',
        Min(c.percentage) 'Min Score',
        Max(c.percentage) 'Max Score',
        Avg(c.percentage) 'Avg Percentage'
FROM    student.studentinfo s
        RIGHT JOIN course.coursegrade c
               ON c.studentno = s.studentno
        JOIN Course.CourseInfo ci
               ON c.CourseNo = ci.CourseNo
GROUP  BY c.courseno, ci.Name
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT c.courseno 'Course No', ci.Name 'Course', Count(s.studentno)…
```

```
SELECT  c.courseno 'Course No',
        ci.Name 'Course',
        Count(s.studentno) 'Total Students',
        Min(c.percentage) 'Min Score',
        Max(c.percentage) 'Max Score',
        Avg(c.percentage) 'Avg Percentage'
FROM    student.studentinfo s
        with(index(PK_StudentInfo))
        RIGHT JOIN course.coursegrade c
               ON c.studentno = s.studentno
        JOIN Course.CourseInfo ci
               ON c.CourseNo = ci.CourseNo
GROUP  BY c.courseno, ci.Name
option(loop join)
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT c.courseno 'Course No', ci.Name 'Course', Count(s.studentno) 'T…
```

~vsE36A.sql - DES...RMRA\coezema (53))*
~vs15C7.sql - DES...RMRA\coezema (52))
~vsC0A2.sql - DES...MRA\coezema (55))*
Results
Messages
Execution plan
Sort
Nested Loops (Left Outer Join)
Hash Match (Inner Join)
Clustered Index Scan (Clu [CourseInfo].[PK_CourseIn
Clustered Index Seek (Clustered) [StudentInfo].[PK_StudentInfo] [s]
Table Scan [CourseGrade] [c]
Nested Loops (Inner Join)
Clustered Index Seek (Clu [CourseInfo].[PK_CourseInf
DESKTOP-CD9RMRA (14.0 RTM) | DESKTOP-CD9RMRA\coezem... | StudentEnrollment | 00:00:01 | 15 rows
DESKTOP-CD9RMRA (14.0 RTM) | DESKTOP-CD9RMRA\coezem... | StudentEnrollment | 00:00:00 | 15 rows

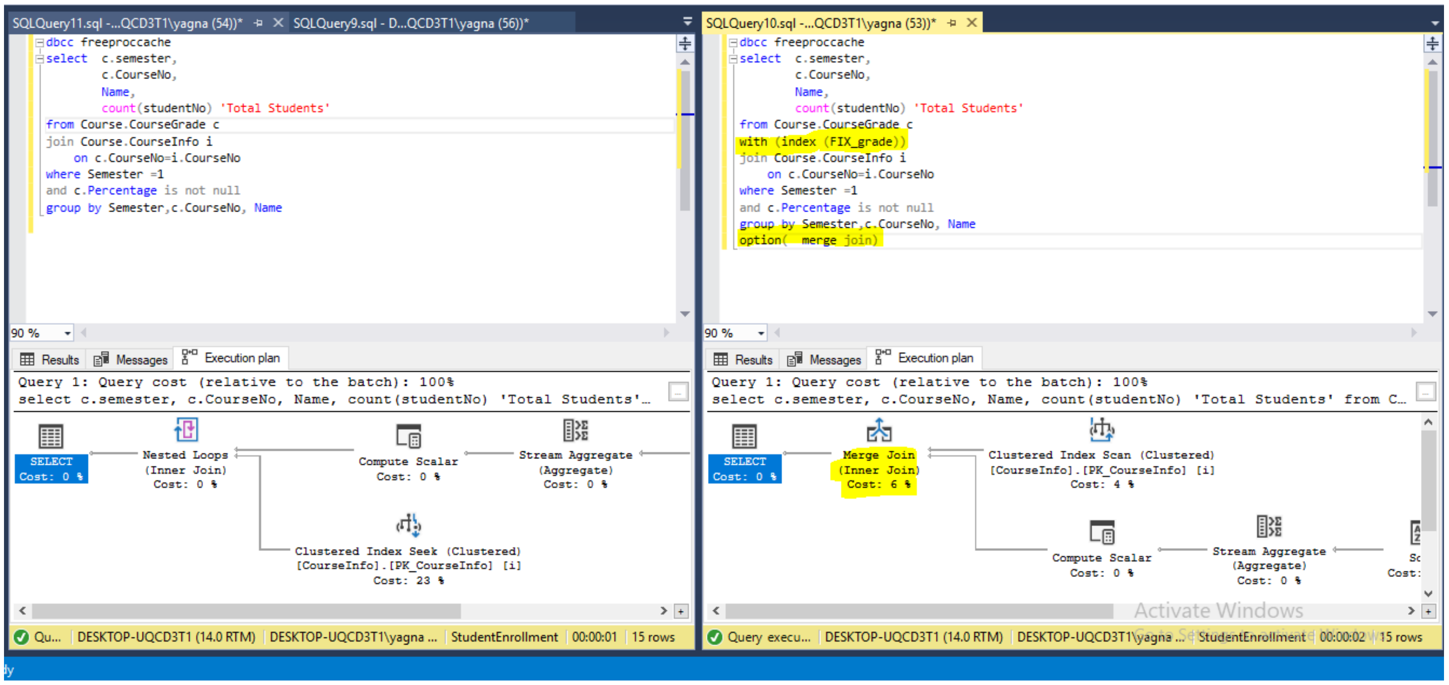
As we can see, by “default” the optimizer opts to use hashing and then develop the counts based on the matched values. We try to use the data from the Clustered Scan in an ordered fashion rather than the unordered Hash Match hint to the query using the loop hint.

We see that the query has introduced 2 nested loops in the execution and has slowed down the efficiency. The initial query cost was 0.049 and using the hints it has gone down to 0.090. Its always best to let the query optimizer do the selection of the method for the run time efficiency.

**Using Merge Join Hint:**

Here we try to optimize the query by using the merge join table query hint. We try to get the number of student in for each course in the first semester. We try to force implement merge join hint in comparison to the default method of the query optimizer. The default method costs 0.021 as compared to 0.0879 for the query using the merge join.

The query optimizer does a pretty good job of selecting the nested loops to optimize the query efficiency. Had this table been using millions of rows, we might have found a significant improvement in performance using the merge join hint in the table.


SQLQuery11.sql -...QCD3T1\yagna (54))*
SQLQuery9.sql - D...QCD3T1\yagna (56))*
SQLQuery10.sql -...QCD3T1\yagna (53))*
90 %
Results
Messages
Execution plan
Activate Windows

```
dbcc freeproccache
select  c.semester,
        c.CourseNo,
        Name,
        count(studentNo) 'Total Students'
from Course.CourseGrade c
join Course.CourseInfo i
    on c.CourseNo=i.CourseNo
where Semester =1
and c.Percentage is not null
group by Semester,c.CourseNo, Name
```

```
Query 1: Query cost (relative to the batch): 100%
select c.semester, c.CourseNo, Name, count(studentNo) 'Total Students'…
```

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Compute Scalar
Cost: 0 %
Stream Aggregate
(Aggregate)
Cost: 0 %
Clustered Index Seek (Clustered)
[CourseInfo].[PK_CourseInfo] [i]
Cost: 23 %

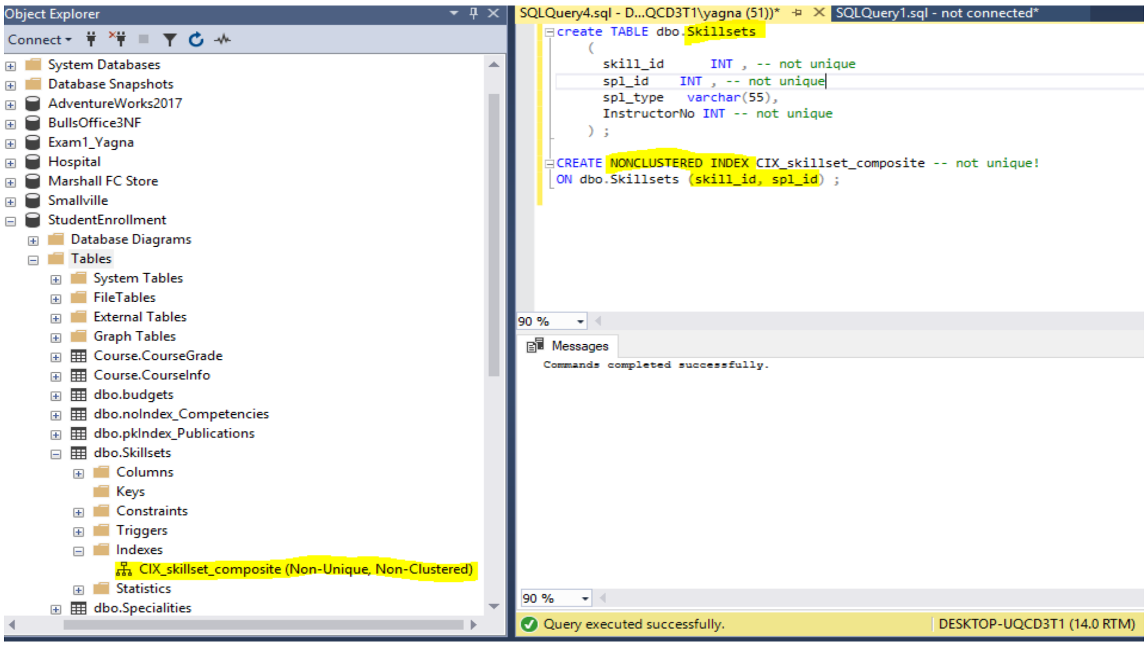
```
Qu...  DESKTOP-UQCD3T1 (14.0 RTM)  DESKTOP-UQCD3T1\yagna ...  StudentEnrollment  00:00:01  15 rows
```

```
dbcc freeproccache
select  c.semester,
        c.CourseNo,
        Name,
        count(studentNo) 'Total Students'
from Course.CourseGrade c
with (index (FIX_grade))
join Course.CourseInfo i
    on c.CourseNo=i.CourseNo
where Semester =1
and c.Percentage is not null
group by Semester,c.CourseNo, Name
option(  merge join)
```

```
Query 1: Query cost (relative to the batch): 100%
select c.semester, c.CourseNo, Name, count(studentNo) 'Total Students' from C…
```

```
Query execu...  DESKTOP-UQCD3T1 (14.0 RTM)  DESKTOP-UQCD3T1\yagna ...  StudentEnrollment  00:00:02  15 rows
```

- **Demonstrate use of: Composite, Covering, Use, Include, With, Filter:**
- **COMPOSITE index using WITH:**
  An index that contains more than one column. In SQL Server, we can include up to 16 columns in an index, as long as the index doesn't exceed the 900-byte limit. Both clustered and non-clustered indexes can be composite indexes.

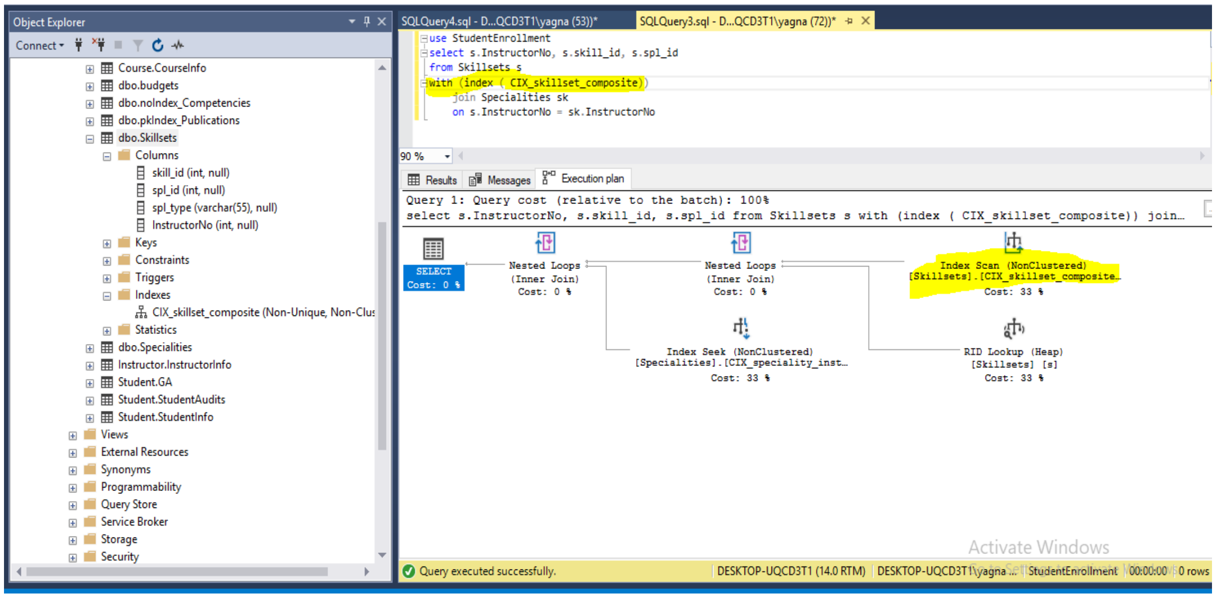


```
create TABLE dbo.Skillsets
    (
      skill_id      INT , -- not unique
      spl_id    INT , -- not unique
      spl_type   varchar(55),
      InstructorNo INT -- not unique
    ) ;

CREATE NONCLUSTERED INDEX CIX_skillset_composite -- not unique!
ON dbo.Skillsets (skill_id, spl_id) ;
```

```
Commands completed successfully.
```

Query executed successfully.    DESKTOP-UQCD3T1 (14.0 RTM)

EXEC PLAN:



```
use StudentEnrollment
select s.InstructorNo, s.skill_id, s.spl_id
from Skillsets s
with (index ( CIX_skillset_composite))
    join Specialities sk
    on s.InstructorNo = sk.InstructorNo
```

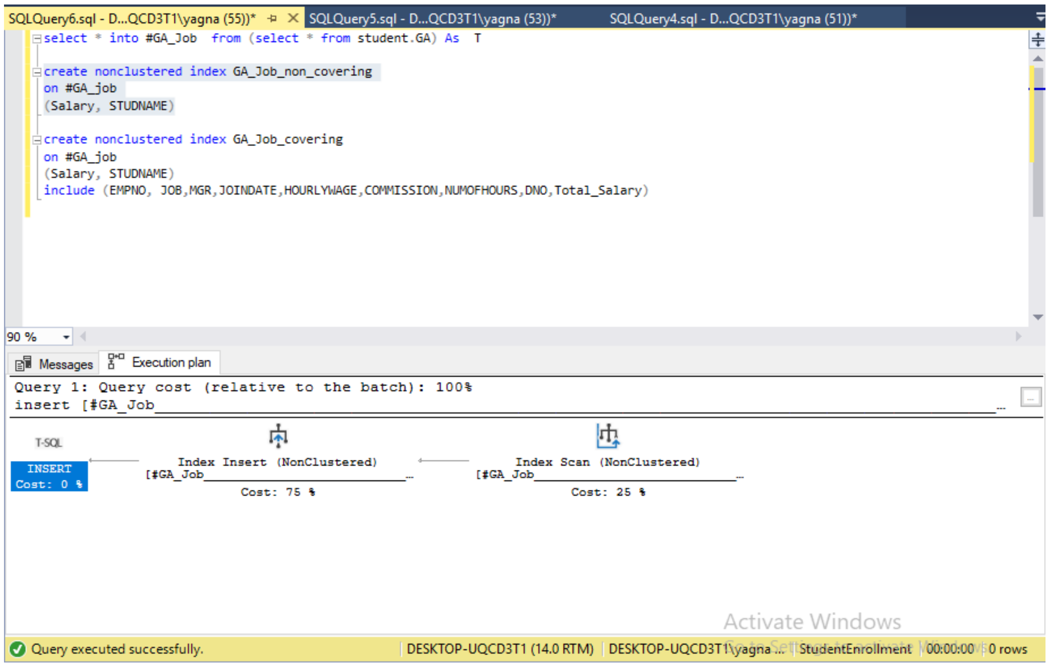
```
Query 1: Query cost (relative to the batch): 100%
select s.InstructorNo, s.skill_id, s.spl_id from Skillsets s with (index ( CIX_skillset_composite)) join…
```

Query executed successfully.    DESKTOP-UQCD3T1 (14.0 RTM)    DESKTOP-UQCD3T1\yagna…    StudentEnrollment    00:00:00    0 rows

We can see that by using a composite index, the index is added to the Skillsets table. The query uses the index scan method to scan through the non-clustered Indexed column for skillset table and uses Index seek method to get the non-clustered indexed for the Specialties ID. The nested loop uses the RID lookup also known as HEAP method to seek the composite index columns in the skillsets table for the select query in the equal 33% cost for all 3.

The query is using HEAP method as the composite indexes are non-clustered and the data stored in the table without a specified order.

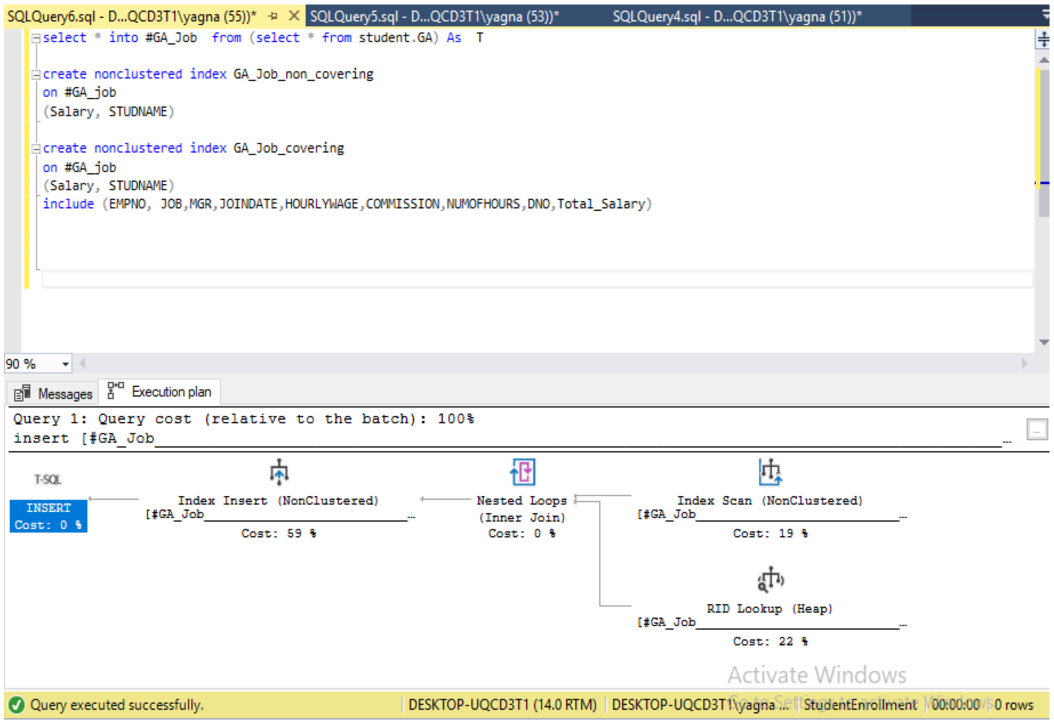
- **<u>Use of COVERING index with INCLUDE:</u>**
  An index that contains all information required to resolve the query is known as a "Covering Index"; it completely covers the query. In this we query, we create a temp table named GA_job and we create a covering and non-covering index in 2 different queries. The covering index query uses the Include syntax to include the columns in the table that will be covered in the indexing (that means all the columns in the table).



```
select * into #GA_Job  from (select * from student.GA) As  T

create nonclustered index GA_Job_non_covering
on #GA_job
(Salary, STUDNAME)

create nonclustered index GA_Job_covering
on #GA_job
(Salary, STUDNAME)
include (EMPNO, JOB,MGR,JOINDATE,HOURLYWAGE,COMMISSION,NUMOFHOURS,DNO,Total_Salary)
```

```
Query 1: Query cost (relative to the batch): 100%
insert [#GA_Job
```

SQLQuery6.sql - D...QCD3T1\yagna (55))*
SQLQuery5.sql - D...QCD3T1\yagna (53))*
SQLQuery4.sql - D...QCD3T1\yagna (51))*
90 %
Messages
Execution plan
T-SQL
INSERT
Cost: 0 %
Index Insert (NonClustered)
[#GA_Job
Cost: 75 %
Index Scan (NonClustered)
[#GA_Job
Cost: 25 %
Activate Windows
Query executed successfully.
DESKTOP-UQCD3T1 (14.0 RTM)
DESKTOP-UQCD3T1\yagna ...
StudentEnrollment
00:00:00
0 rows



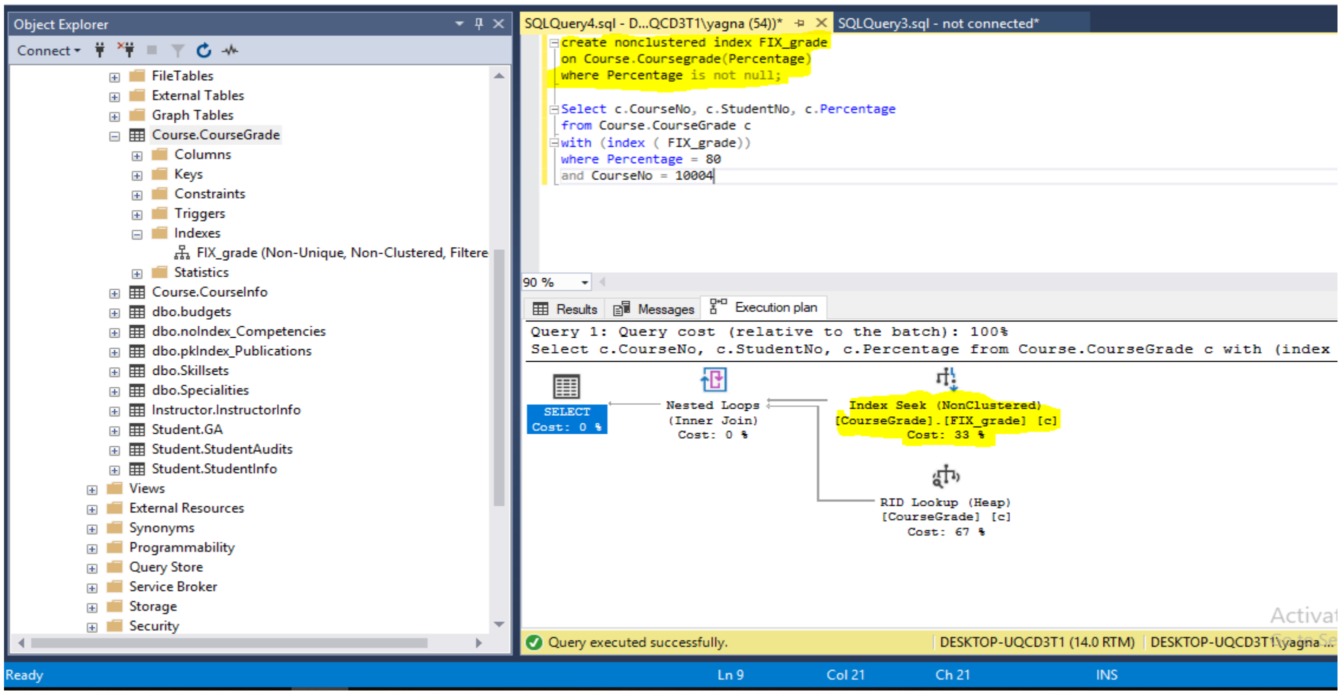
```
select * into #GA_Job  from (select * from student.GA) As  T

create nonclustered index GA_Job_non_covering
on #GA_job
(Salary, STUDNAME)

create nonclustered index GA_Job_covering
on #GA_job
(Salary, STUDNAME)
include (EMPNO, JOB,MGR,JOINDATE,HOURLYWAGE,COMMISSION,NUMOFHOURS,DNO,Total_Salary)
```

```
Query 1: Query cost (relative to the batch): 100%
insert [#GA_Job
```

SQLQuery6.sql - D...QCD3T1\yagna (55))*
SQLQuery5.sql - D...QCD3T1\yagna (53))*
SQLQuery4.sql - D...QCD3T1\yagna (51))*
90 %
Messages
Execution plan
T-SQL
INSERT
Cost: 0 %
Index Insert (NonClustered)
[#GA_Job
Cost: 59 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Scan (NonClustered)
[#GA_Job
Cost: 19 %
RID Lookup (Heap)
[#GA_Job
Cost: 22 %
Activate Windows
Query executed successfully.
DESKTOP-UQCD3T1 (14.0 RTM)
DESKTOP-UQCD3T1\yagna ...
StudentEnrollment
00:00:00
0 rows

The biggest benefit of covering index is that it contains all the fields required by query, it greatly improves the query performance and best results are achieved by placing fields required in the join criteria into the index key and placing fields required in the SELECT list into the INCLUDE part of the covering index. We can see that there is a greater benefit in using the covering index in the GA temp table. We see that the query cost is only 19% for the index scan for the non-clustered index and the Heap method cost is only 22% as compared to 25% for the initial index scan in the non-covering index.

- **Filtered Index:**
  Filtered indexes are non-clustered indexes that have the addition of a WHERE clause. Although the WHERE clause in a filtered index allows only simple predicates, it provides notable improvements over a traditional non-clustered index. This allows the index to target specific data values – only records with certain values will be added to the index – resulting in a much smaller, more accurate, and more efficient index.
  Here, we add a filtered index to the percentage column of the coursegrade table. The query is run with the select statement with selected percentage and the courseNo forced with the Filtered Index using the WITH clause.



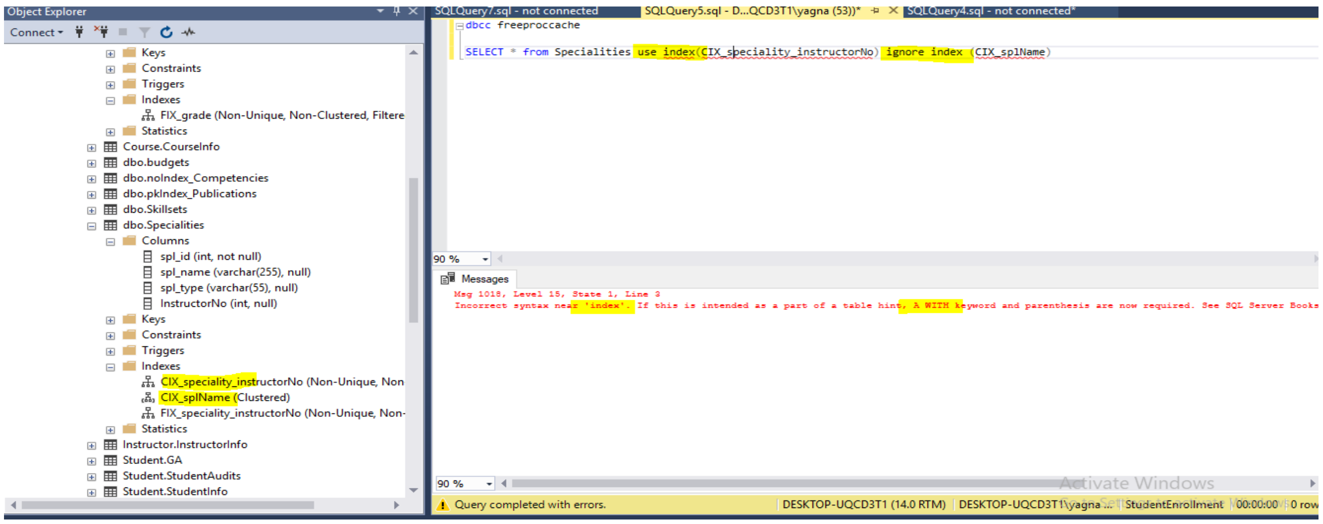
```
create nonclustered index FIX_grade
on Course.Coursegrade(Percentage)
where Percentage is not null;

Select c.CourseNo, c.StudentNo, c.Percentage
from Course.CourseGrade c
with (index ( FIX_grade))
where Percentage = 80
and CourseNo = 10004
```

```
Query 1: Query cost (relative to the batch): 100%
Select c.CourseNo, c.StudentNo, c.Percentage from Course.CourseGrade c with (index
```

We can see that the index seek method for the filtered index costs only 33% and the rest of it goes to the HEAP method for the non-clustered indexes. If we look at the current execution plan for this query, we see that there are no existing indexes that could be used for an index seek to locate the matching records efficiently, and that a clustered index scan was performed instead. An index seek operation happens when the query engine can use an appropriate index to find the exact location of the requested data, using the b-tree method.

## Use Index:

We tried to implement USE clause in the index query but SQL Server showed an error on how it can't be used in querying the indexed columns.



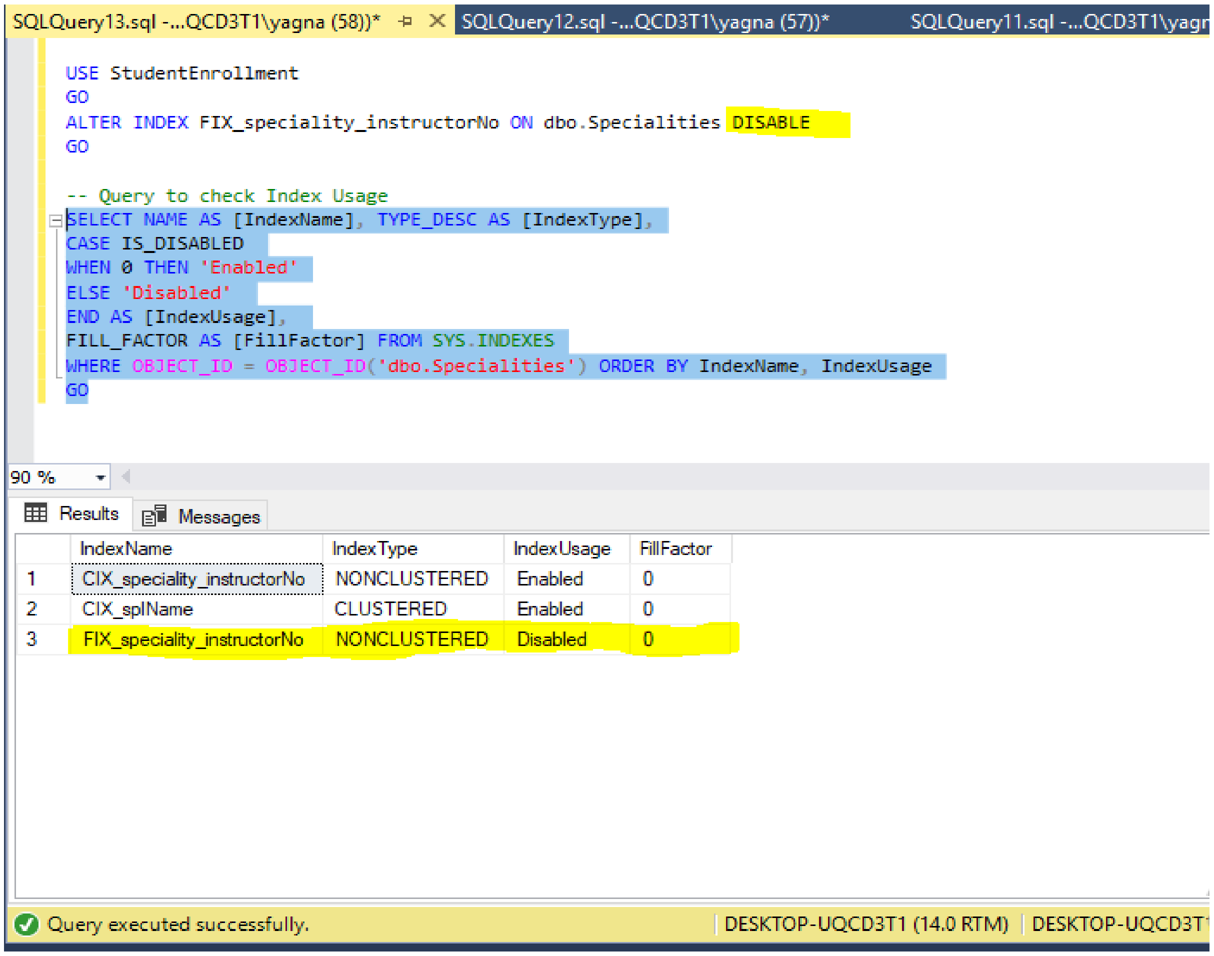
```
dbcc freeproccache

SELECT * from Specialities use index(CIX_speciality_instructorNo) ignore index (CIX_splName)
```

```
Msg 1018, Level 15, State 1, Line 3
Incorrect syntax near 'index'. If this is intended as a part of a table hint, A WITH keyword and parenthesis are now required. See SQL Server Books
```

CIX_speciality_instructorNo (Non-Unique, Non
CIX_splName (Clustered)
FIX_speciality_instructorNo (Non-Unique, Non-
Query completed with errors.

## Demonstrate use of: disable, rebuild, drop:

In order to improve our applications and our databases, they will need to change over time. The structure of the database changes, the structure of the tables changes, the data in the tables change, the application changes, the queries against the data change. Indexes that once helped performance now just bloat the database and cause extra work for inserts, updates, and deletes. When index needs change and we want to test how removing an index will affect performance, we have two options – we can disable or drop the index.



SQLQuery13.sql -...QCD3T1\yagna (58))* | SQLQuery12.sql -...QCD3T1\yagna (57))* | SQLQuery11.sql -...QCD3T1\yagn

```
USE StudentEnrollment
GO
ALTER INDEX FIX_speciality_instructorNo ON dbo.Specialities DISABLE
GO

-- Query to check Index Usage
SELECT NAME AS [IndexName], TYPE_DESC AS [IndexType],
CASE IS_DISABLED
WHEN 0 THEN 'Enabled'
ELSE 'Disabled'
END AS [IndexUsage],
FILL_FACTOR AS [FillFactor] FROM SYS.INDEXES
WHERE OBJECT_ID = OBJECT_ID('dbo.Specialities') ORDER BY IndexName, IndexUsage
GO
```

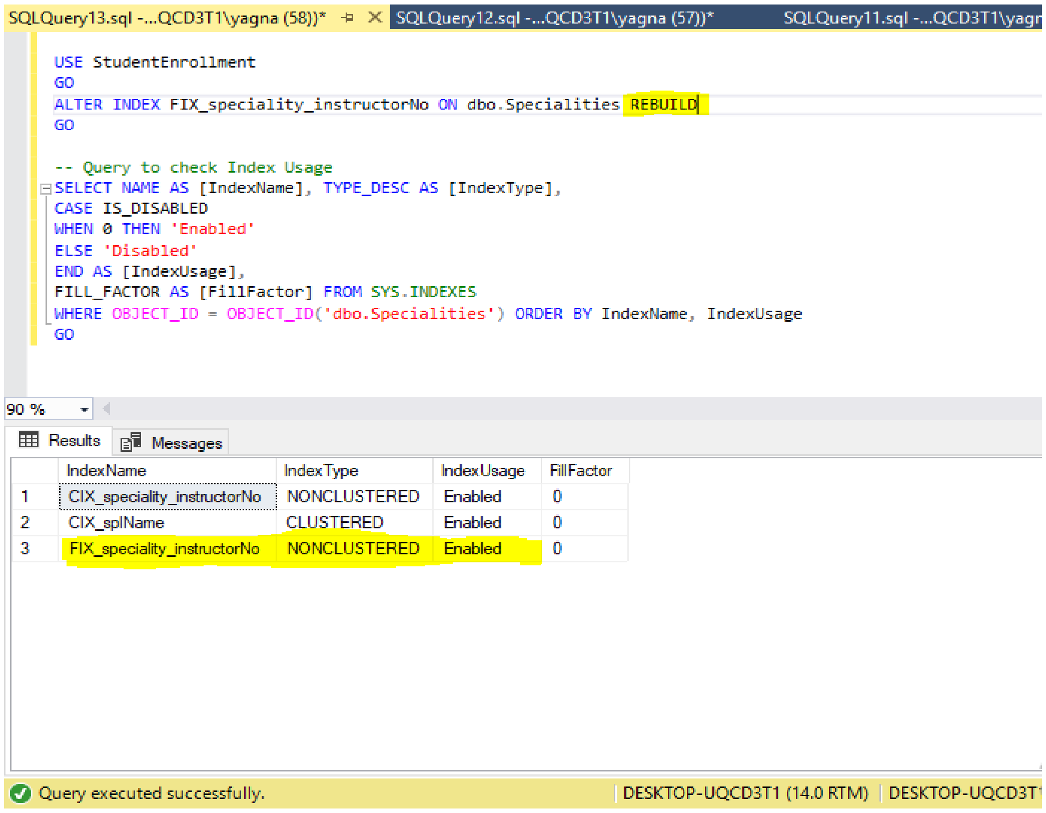
| | IndexName | IndexType | IndexUsage | FillFactor |
|---|---|---|---|---|
| 1 | CIX_speciality_instructorNo | NONCLUSTERED | Enabled | 0 |
| 2 | CIX_splName | CLUSTERED | Enabled | 0 |
| 3 | FIX_speciality_instructorNo | NONCLUSTERED | Disabled | 0 |

Query executed successfully. | DESKTOP-UQCD3T1 (14.0 RTM) | DESKTOP-UQCD3T

We see in the above query that the filtered non cluster index has been altered to disable in the first part and we see the status of the indexes in the table. We can see that the index usage has been  disabled in the table below.

Disabling a non-clustered index will deallocate the index pages – the space is freed in the database. Disabling a clustered index has additional effects. The data in the table still exists but will be inaccessible for anything other than a drop or rebuild operation. All related non-clustered indexes and views are also unavailable. Foreign key constraints that reference the table are disabled. Queries against the table will fail.

To enable the index we use the REBUILD command like this:


SQLQuery13.sql -...QCD3T1\yagna (58))* SQLQuery12.sql -...QCD3T1\yagna (57))* SQLQuery11.sql -...QCD3T1\yagr

```
USE StudentEnrollment
GO
ALTER INDEX FIX_speciality_instructorNo ON dbo.Specialities REBUILD
GO

-- Query to check Index Usage
SELECT NAME AS [IndexName], TYPE_DESC AS [IndexType],
CASE IS_DISABLED
WHEN 0 THEN 'Enabled'
ELSE 'Disabled'
END AS [IndexUsage],
FILL_FACTOR AS [FillFactor] FROM SYS.INDEXES
WHERE OBJECT_ID = OBJECT_ID('dbo.Specialities') ORDER BY IndexName, IndexUsage
GO
```

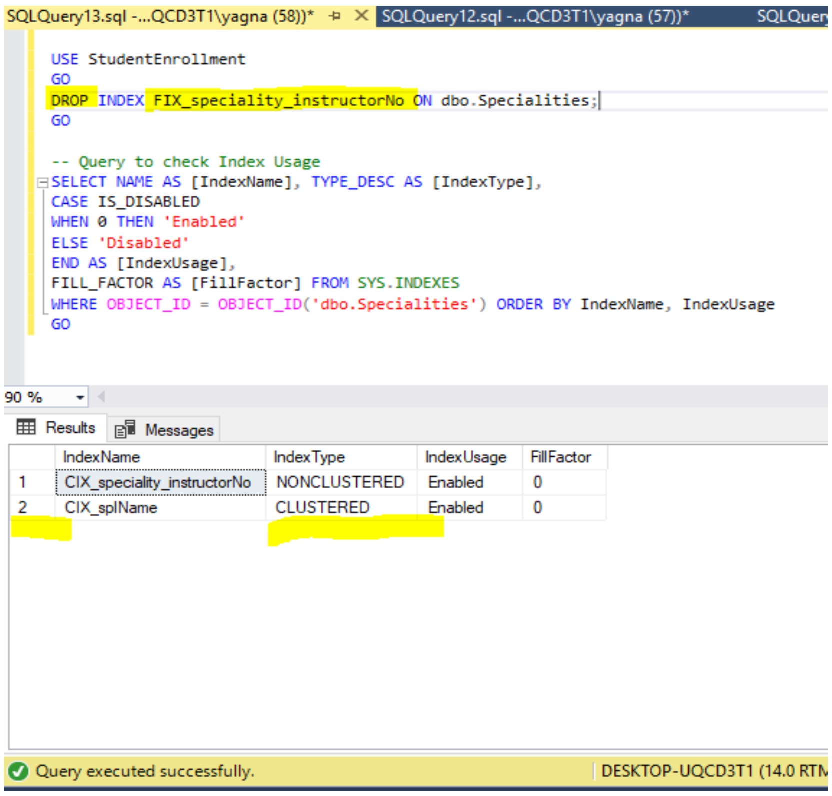
90 %

Results | Messages

| | IndexName | IndexType | IndexUsage | FillFactor |
|---|---|---|---|---|
| 1 | CIX_speciality_instructorNo | NONCLUSTERED | Enabled | 0 |
| 2 | CIX_splName | CLUSTERED | Enabled | 0 |
| 3 | FIX_speciality_instructorNo | NONCLUSTERED | Enabled | 0 |

Query executed successfully. DESKTOP-UQCD3T1 (14.0 RTM) DESKTOP-UQCD3T

To drop a clustered or non-clustered index, issue a DROP INDEX command. When we do this, the metadata, statistics, and index pages are removed. If we drop a clustered index, the table will become a heap. Once an index has been dropped, it can't be rebuilt – it must be created again.


SQLQuery13.sql -...QCD3T1\yagna (58))* SQLQuery12.sql -...QCD3T1\yagna (57))* SQLQuer

```
USE StudentEnrollment
GO
DROP INDEX FIX_speciality_instructorNo ON dbo.Specialities;
GO

-- Query to check Index Usage
SELECT NAME AS [IndexName], TYPE_DESC AS [IndexType],
CASE IS_DISABLED
WHEN 0 THEN 'Enabled'
ELSE 'Disabled'
END AS [IndexUsage],
FILL_FACTOR AS [FillFactor] FROM SYS.INDEXES
WHERE OBJECT_ID = OBJECT_ID('dbo.Specialities') ORDER BY IndexName, IndexUsage
GO
```

90 %

Results | Messages

| | IndexName | IndexType | IndexUsage | FillFactor |
|---|---|---|---|---|
| 1 | CIX_speciality_instructorNo | NONCLUSTERED | Enabled | 0 |
| 2 | CIX_splName | CLUSTERED | Enabled | 0 |

Query executed successfully. DESKTOP-UQCD3T1 (14.0 RTM

We see that the index has been dropped from the table is nowhere to be found on the table in the index stats.